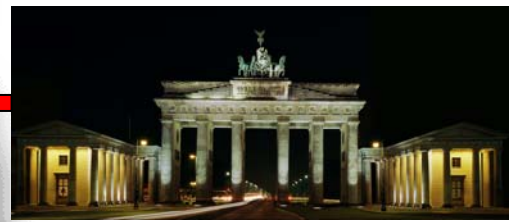
Design of Application Specific Processor Architectures

Rainer Leupers
RWTH Aachen University
Software for Systems on Silicon
leupers@iss.rwth-aachen.de

Overview: Geography

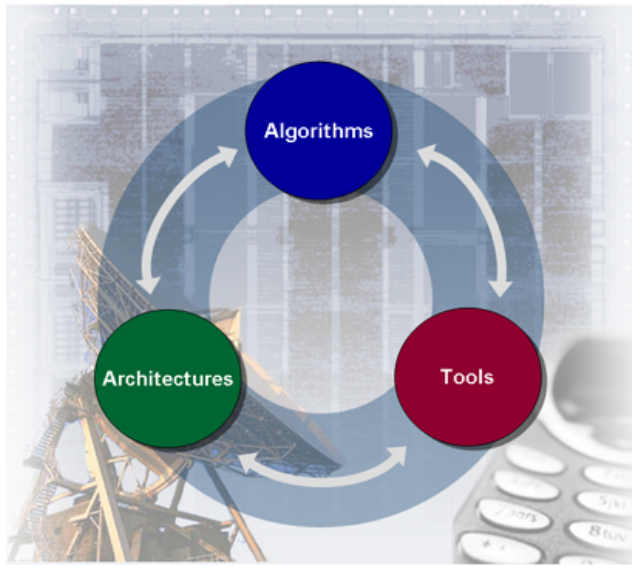


Aachen

Berlin

About ISS

- RWTH Aachen is a top-ranked technical university in Germany
- ISS institute
  - 3 professors (Meyr, Ascheid, Leupers)
  - 18 Ph.D. students
  - 5 staff
- Research on wireless communication systems
  - tight industry cooperations
  - origin of several EDA spin-off companies (e.g. Cadis, Axys, LISATek)
  - SSS group focuses on embedded processor design tools



Overview

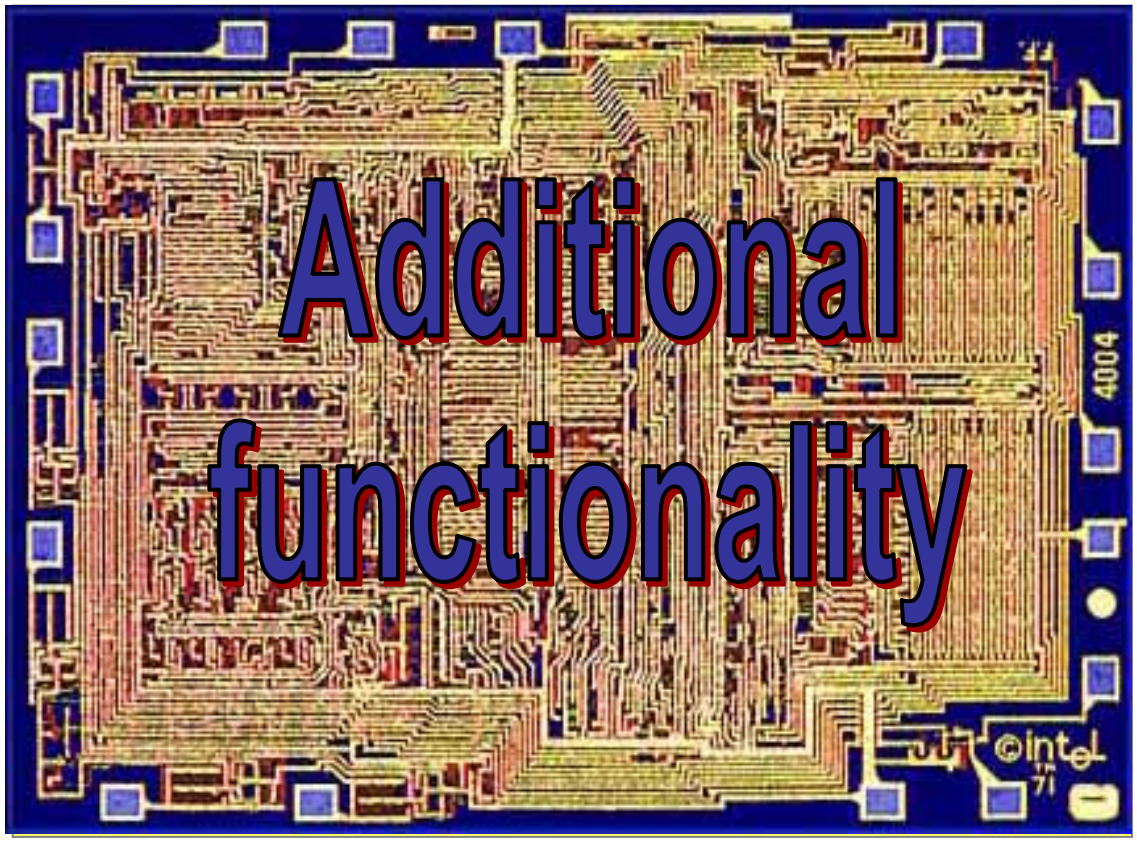
1. Introduction
2. ASIP design methodologies
3. Software tools
4. ASIP architecture design
5. Case study
6. Advanced research topics

1. Introduction

Embedded system design automation

- Embedded systems
  - Special-purpose electronic devices
  - Very different from desktop computers
- Strength of European IT market
  - Telecom, consumer, automotive, medical, ...
  - Siemens, Nokia, Bosch, Infineon, ...
- New design requirements
  - Low NRE cost, high efficiency requirements
  - Real-time operation, dependability
  - Keep pace with Moore´s Law

What to do with chip area ?



Example: wireless multimedia terminals

- Multistandard radio
  - UMTS
  - GSM/GPRS/EDGE
  - WLAN
  - Bluetooth
  - UWB
  - …
- Multimedia standards
  - MPEG-4
  - MP3
  - AAC
  - GPS
  - DVB-H
  - …

Key issues:

- Time to market (≤ 12 months)
- Flexibility (ongoing standard updates)
- Efficiency (battery operation)

Application specific processors (ASIPs)

„As the performance of conventional microprocessors improves, they first meet and then exceed the requirements of most computing applications. Initially, performance is key. But eventually, other factors, like customization, become more important to the customer...“

[M.J. Bass, C.M. Christensen: The Future of the Microprocessor Business, IEEE Spectrum 2002]

design budget = (semiconductor revenue) × (% for R&D)

growth ≈ 15% ≈ 10%

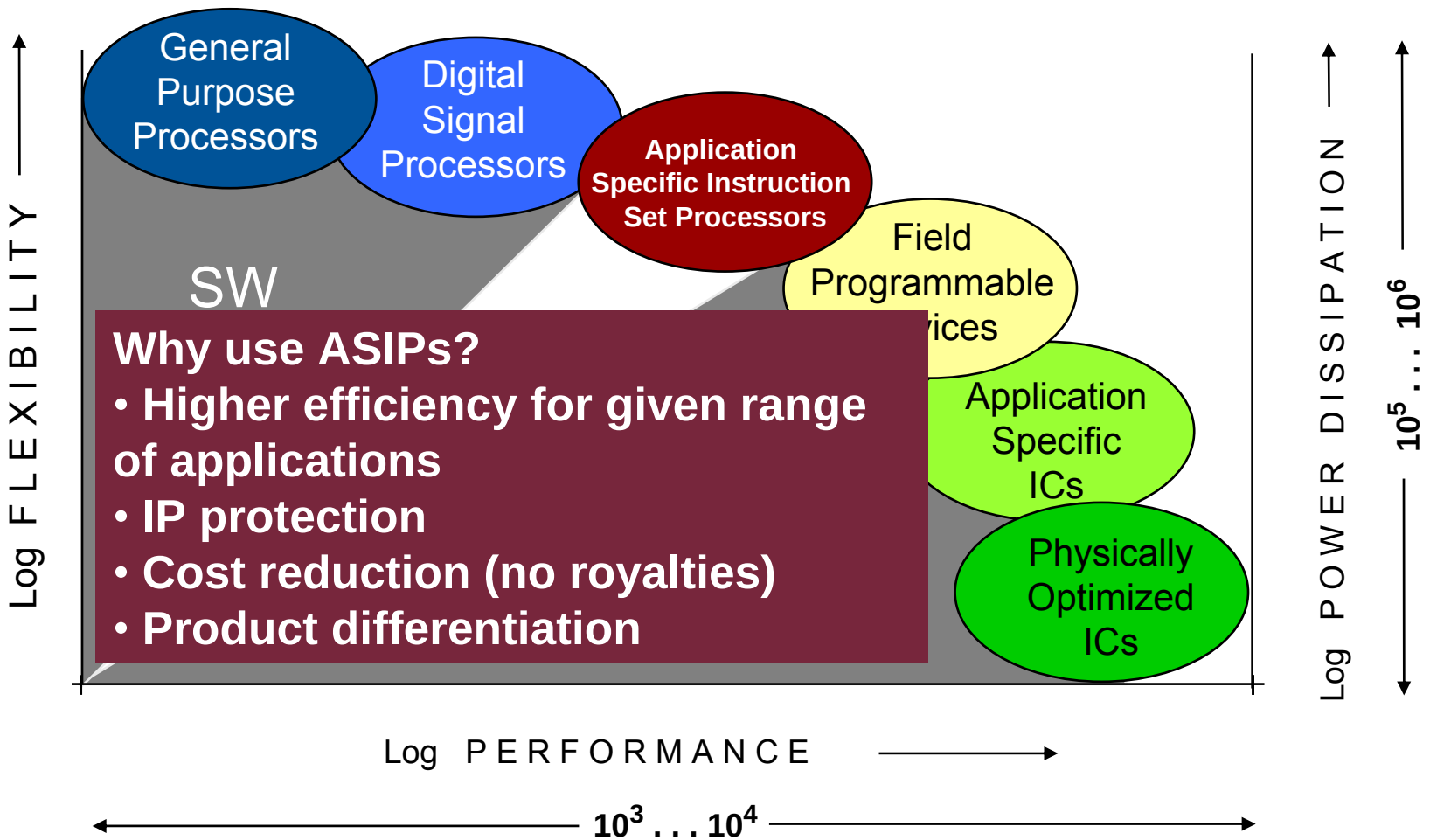
IC designs = (design budget) / (design cost per IC)

growth ≈ 15% growth ≈ 50-100%

[Keutzer05]

→ ***Customizable application specific processors as reusable, programmable platforms***

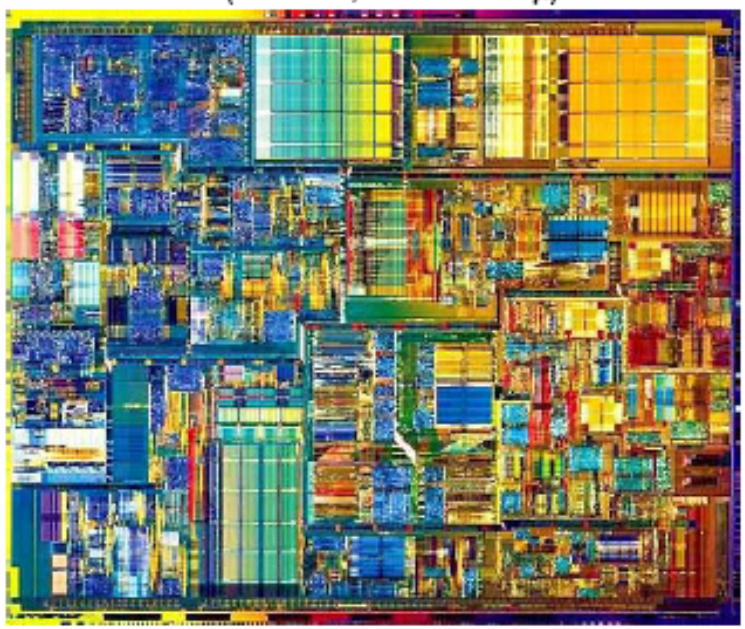
Efficiency and flexibility



Source: T.Noll, RWTH Aachen

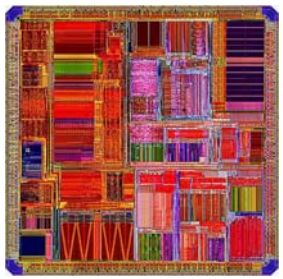
Standard-CPU vs. ASIP

Intel
Pentium 4
(145mm^2, 50W in 0.13µ)

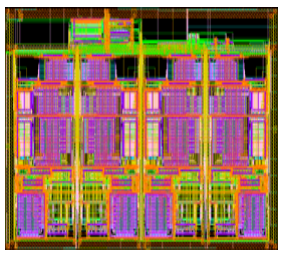


2. ASIP design methodologies

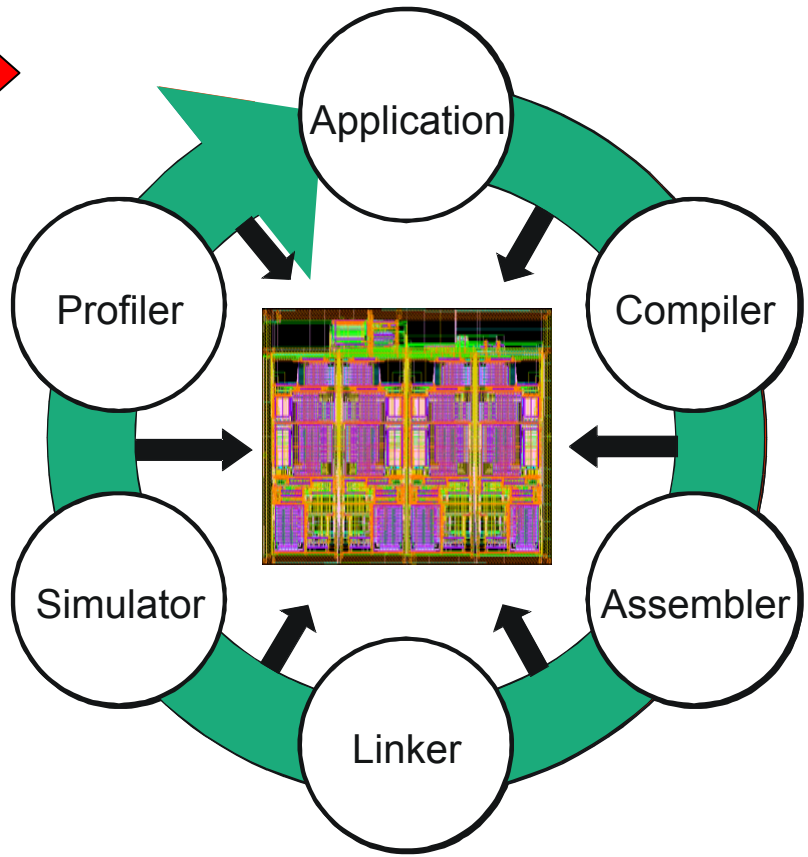
ASIP architecture exploration



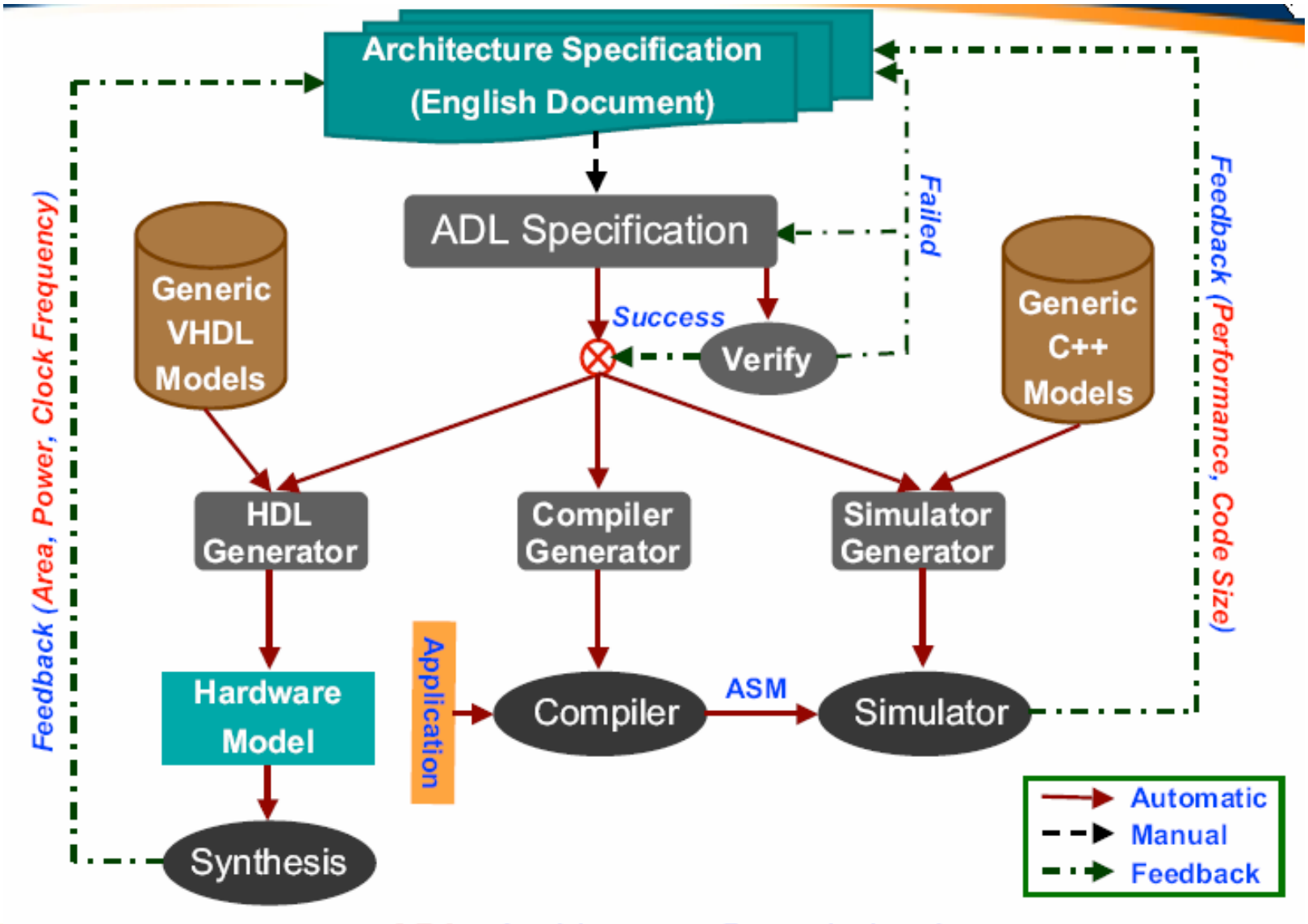
initial processor architecture



optimized processor architecture

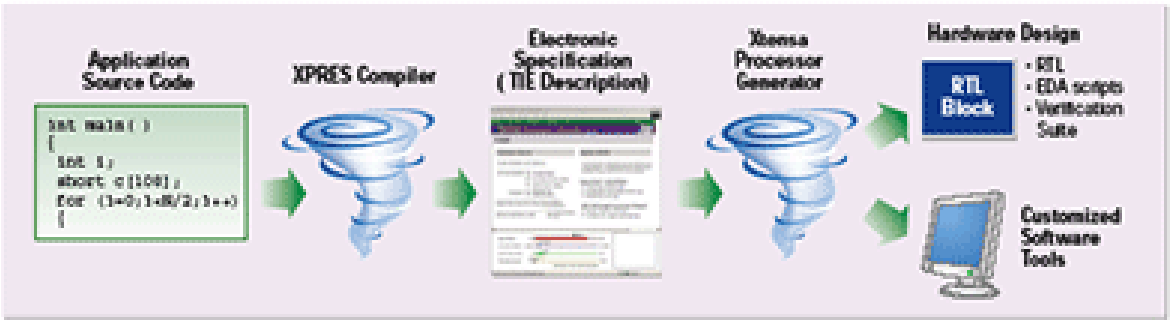


Expression (UC Irvine)



Tensilica Xtensa/XPRES

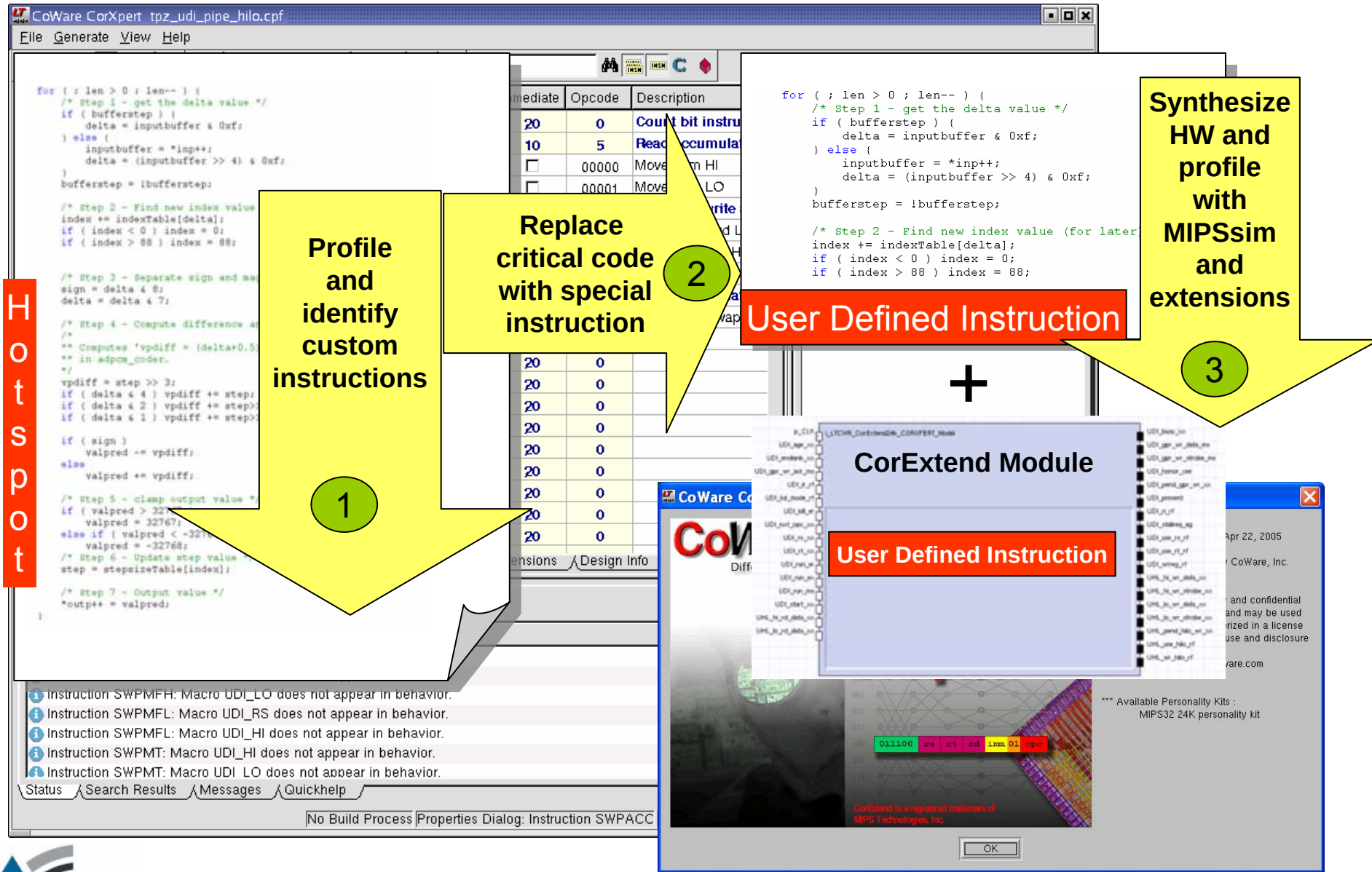
From an ANSI C / C++ application the XPRES Compiler generates an optimized set of processor extensions …



… that is reusable over a range of similar application software code.

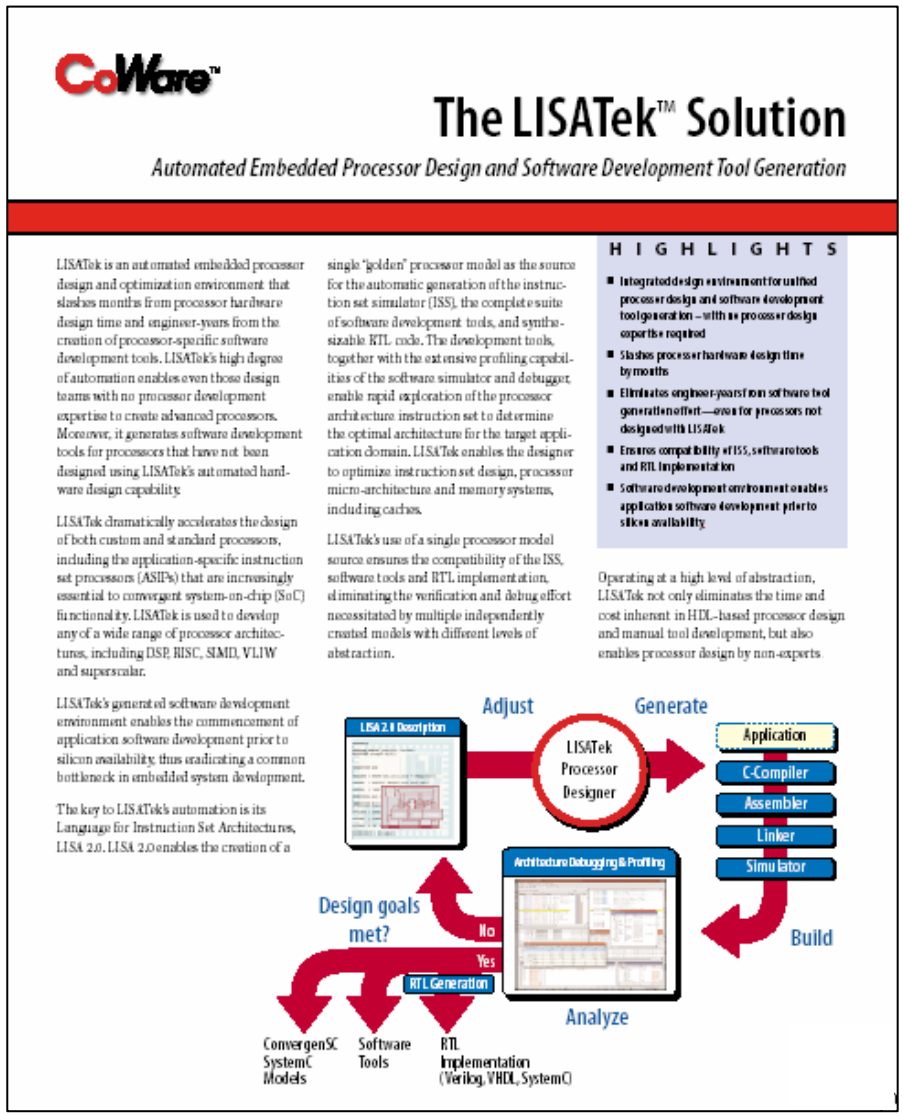
Source: Tensilica Inc.

MIPS CorXtend/CoWare CorXpert



CoWare LISATek ASIP architecture exploration

- Integrated embedded processor development environment
- Unified processor model in LISA 2.0 architecture description language (ADL)
- Automatic generation of:
  - SW tools
  - HW models



CoWare™

The LISATek™ Solution

Automated Embedded Processor Design and Software Development Tool Generation

LISATek is an automated embedded processor design and optimization environment that slashes months from processor hardware design time and engineer-years from the creation of processor-specific software development tools. LISATek's high degree of automation enables even those design teams with no processor development expertise to create advanced processors. Moreover, it generates software development tools for processors that have not been designed using LISATek's automated hardware design capability.

LISATek dramatically accelerates the design of both custom and standard processors, including the application-specific instruction set processors (ASIPs) that are increasingly essential to convergent system-on-chip (SoC) functionality. LISATek is used to develop any of a wide range of processor architectures, including DSP, RISC, SIMD, VLIW and superscalar.

LISATek's generated software development environment enables the commencement of application software development prior to silicon availability, thus eradicating a common bottleneck in embedded system development.

The key to LISATek's automation is its Language for Instruction Set Architectures, LISA 2.0. LISA 2.0 enables the creation of a single "golden" processor model as the source for the automatic generation of the instruction set simulator (ISS), the complete suite of software development tools, and synthesizable RTL code. The development tools, together with the extensive profiling capabilities of the software simulator and debugger, enable rapid exploration of the processor architecture instruction set to determine the optimal architecture for the target application domain. LISATek enables the designer to optimize instruction set design, processor micro-architecture and memory systems, including caches.

LISATek's use of a single processor model source ensures the compatibility of the ISS, software tools and RTL implementation, eliminating the verification and debug effort necessitated by multiple independently created models with different levels of abstraction.

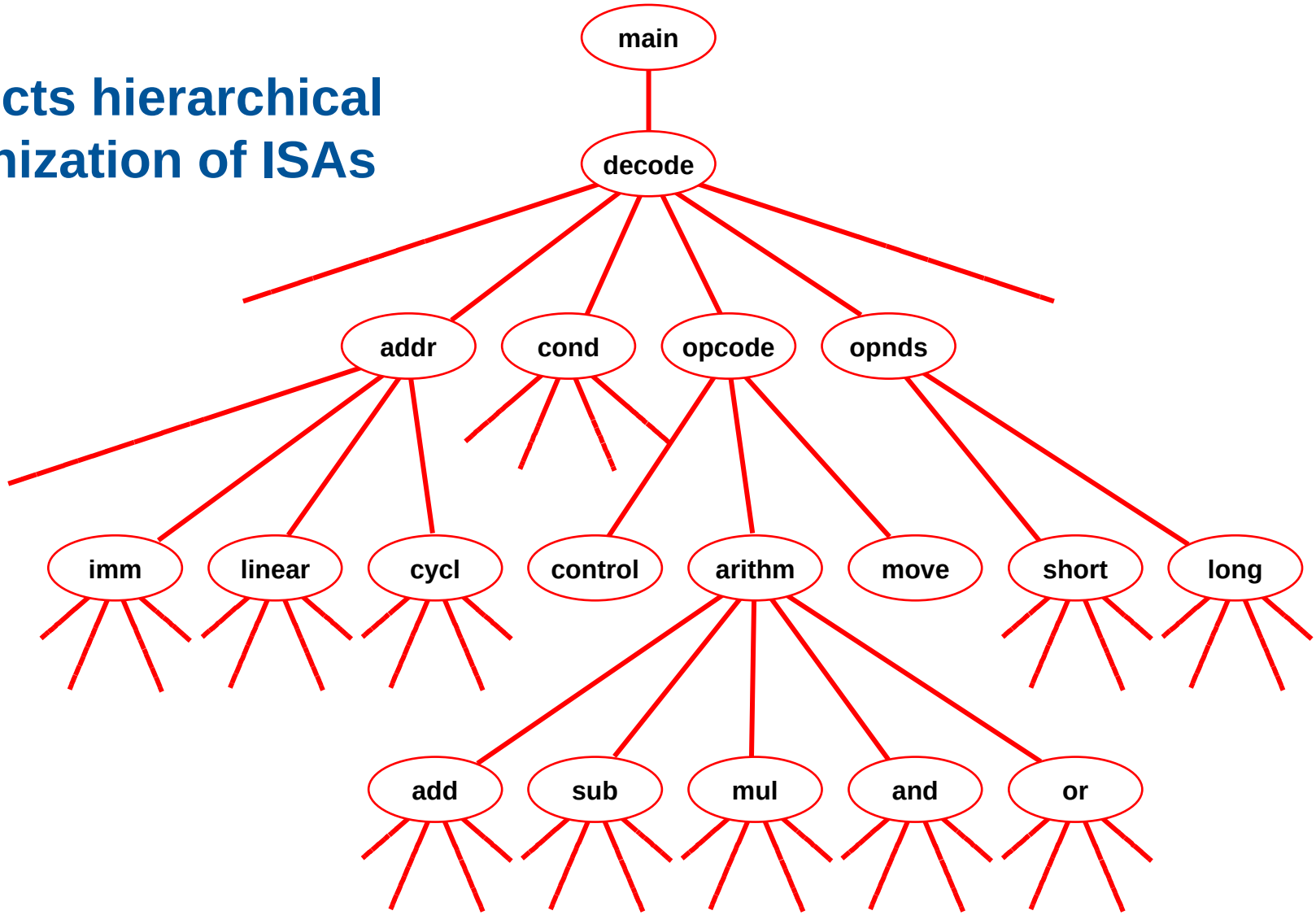
HIGHLIGHTS

- Integrated design environment for unified processor design and software development tool generation – with no processor design expertise required
- Slashes processor hardware design time by months
- Eliminates engineer-years from software tool generation effort—even for processors not designed with LISATek
- Ensures compatibility of ISS, software tools and RTL implementation
- Software development environment enables application software development prior to silicon availability

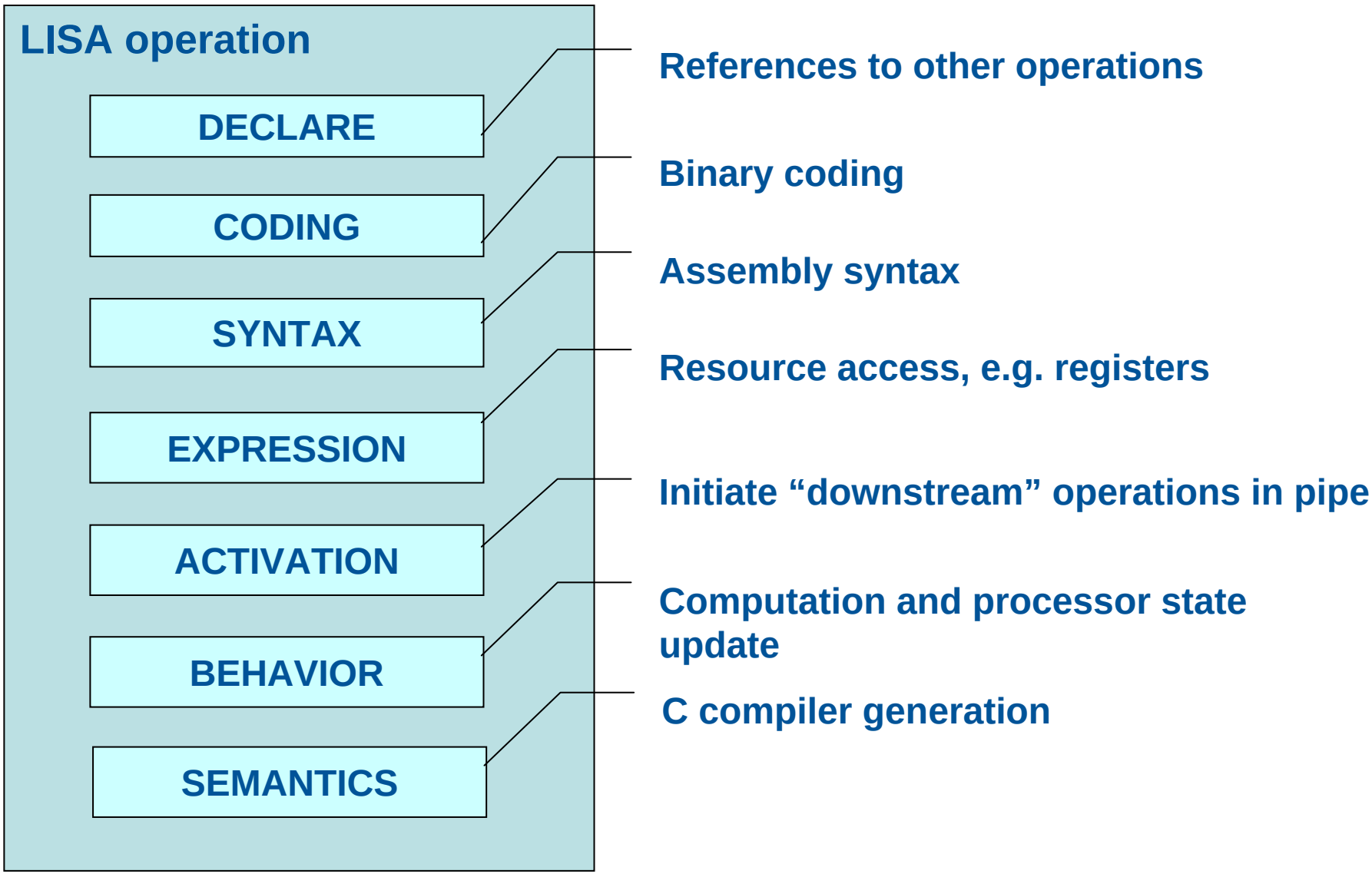
Operating at a high level of abstraction, LISATek not only eliminates the time and cost inherent in HDL-based processor design and manual tool development, but also enables processor design by non-experts.

LISA operation hierarchy

Reflects hierarchical organization of ISAs



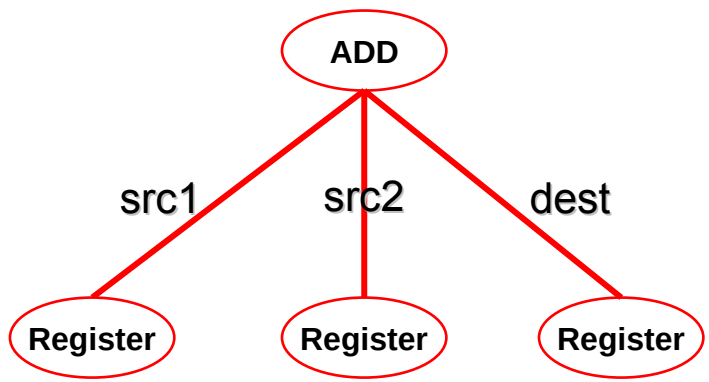
LISA operations structure



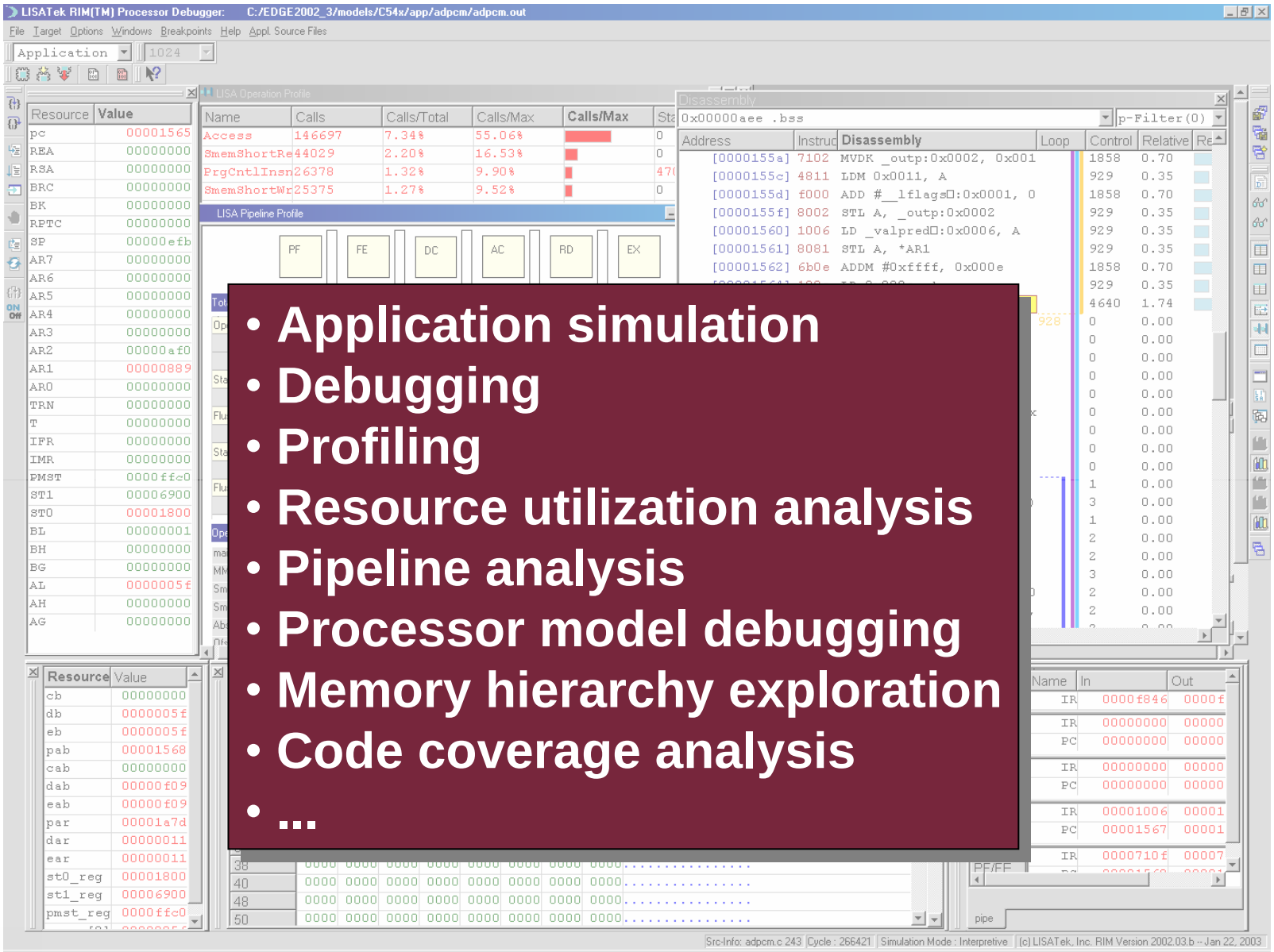
LISA operation example

```
OPERATION ADD
    {
        DECLARE
        {
                GROUP src1, src2, dest = { Register }
        }
        CODING { 0b1011 src1 src2 dest }

        SYNTAX { “ADD” dest “,” src1 “,” src2 }

        BEHAVIOR { dest = src1 + src2; }
    }
OPERATION Register
    {
        DECLARE
        {
                LABEL  index;
        }

        CODING { index }

        SYNTAX { “R”  index }
        EXPRESSION{ R[index] }
    }
```

C/C++ Code



Exploration/debugger GUI

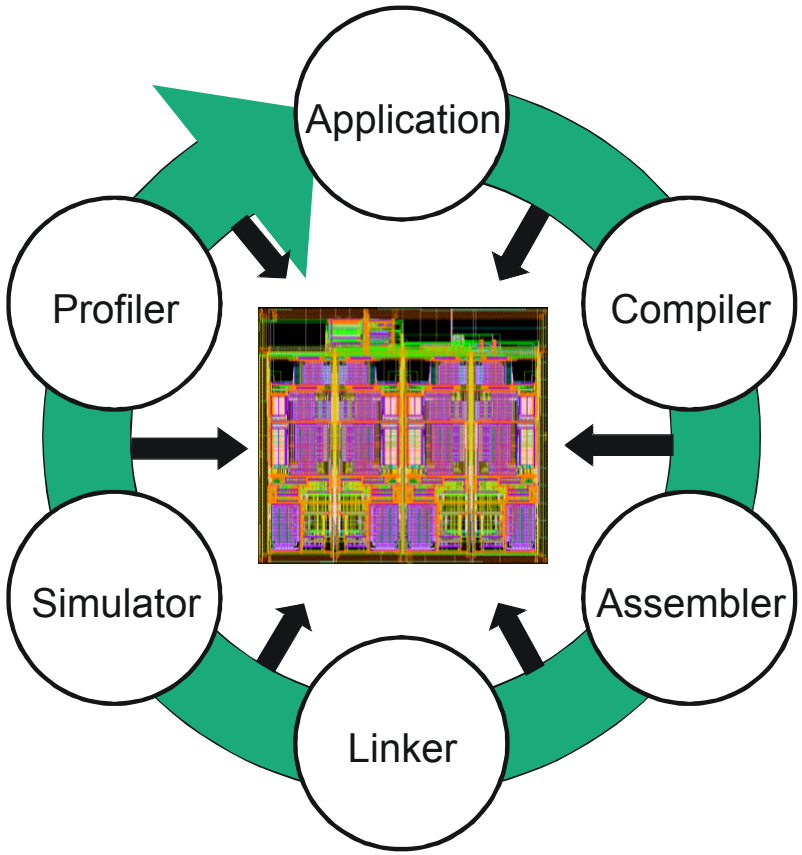


Some available LISA 2.0 models

- DSP:
  - Texas Instruments TMS320C54x
  - Analog Devices ADSP21xx
  - Motorola 56000
- RISC:
  - MIPS32 4K
  - ESA LEON SPARC 8
  - ARM7100
  - ARM926
- VLIW:
  - Texas Instruments TMS320C6x
  - STMicroelectronics ST220
- µC:
  - MHS80C51
- ASIP:
  - Infineon PP32 NPU
  - Infineon ICore
  - MorphICs DSP

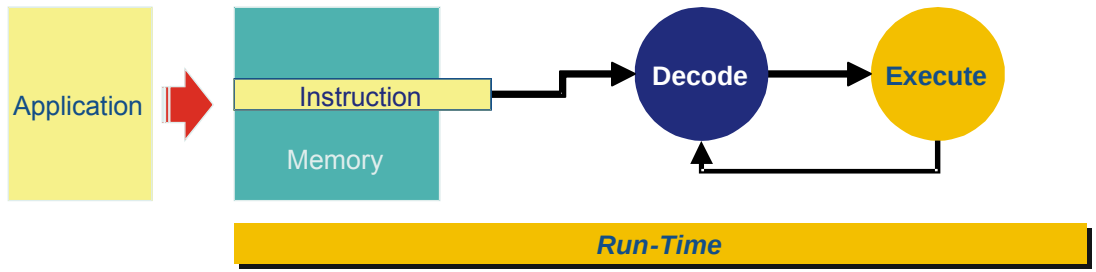
3. Software tools

Tools generated from processor ADL model



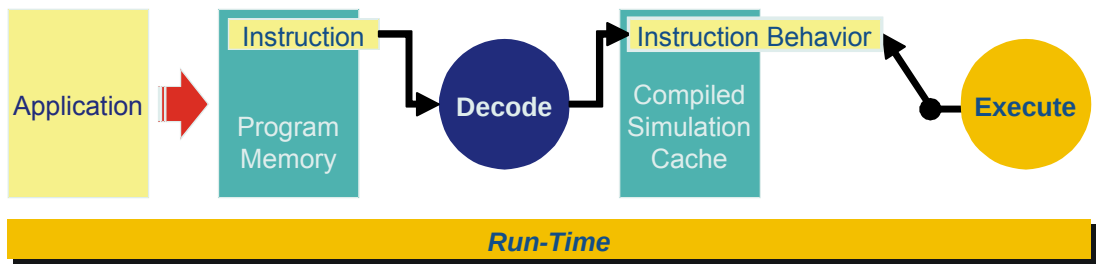
Instruction set simulation

Interpretive:
- flexible
- slow (~ 100 KIPS)



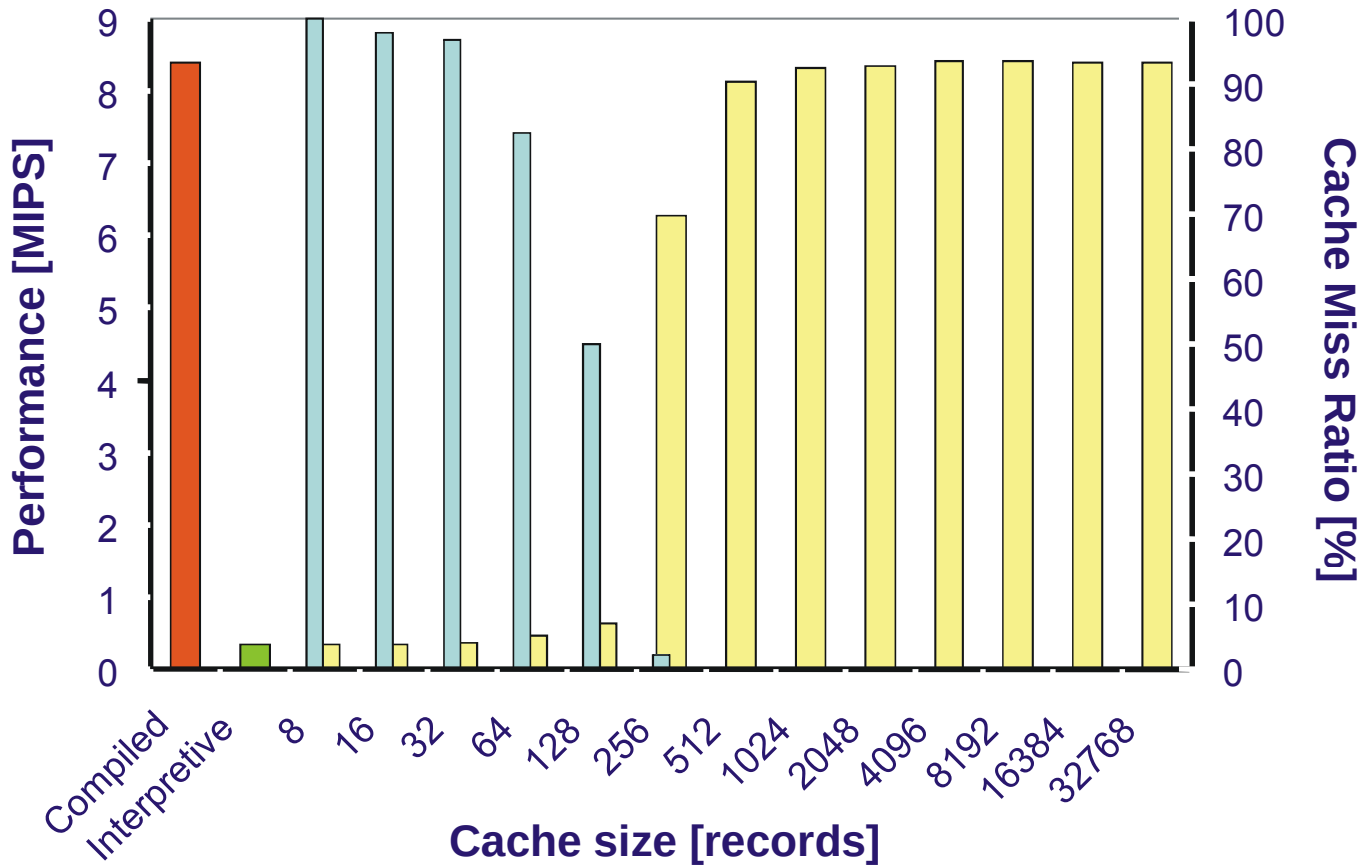
Compiled:
- fast (> 10 MIPS)
- inflexible
- high memory consumption

JIT-CCS™:
- „just-in-time“ compiled
- SW simulation cache
- fast and flexible



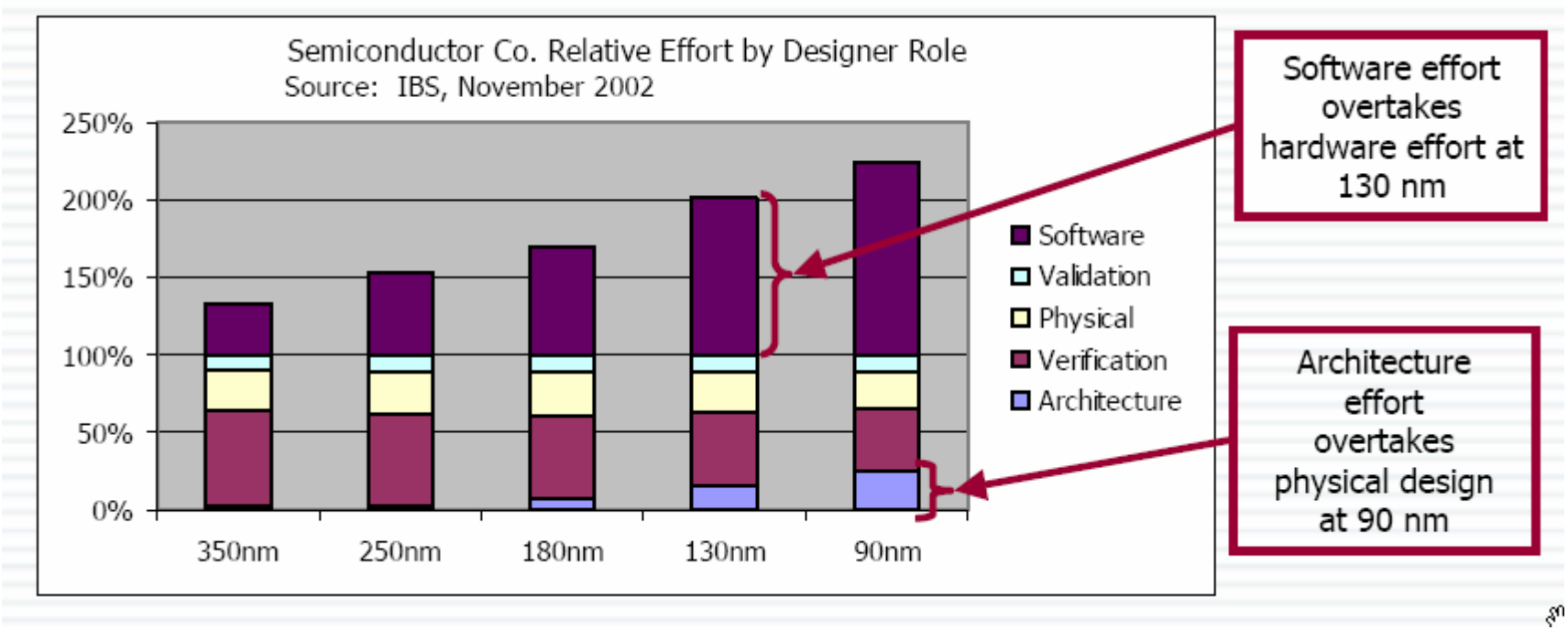
JIT-CC simulation performance

- Dependent on simulation cache size
- 95% of compiled simulation performance @ 4096 cache blocks (10% memory consumption of compiled sim.)
- Example: ST200 VLIW DSP



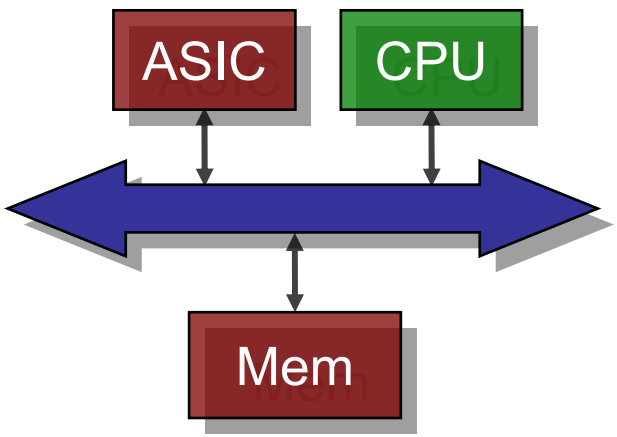
Why care about C compilers?

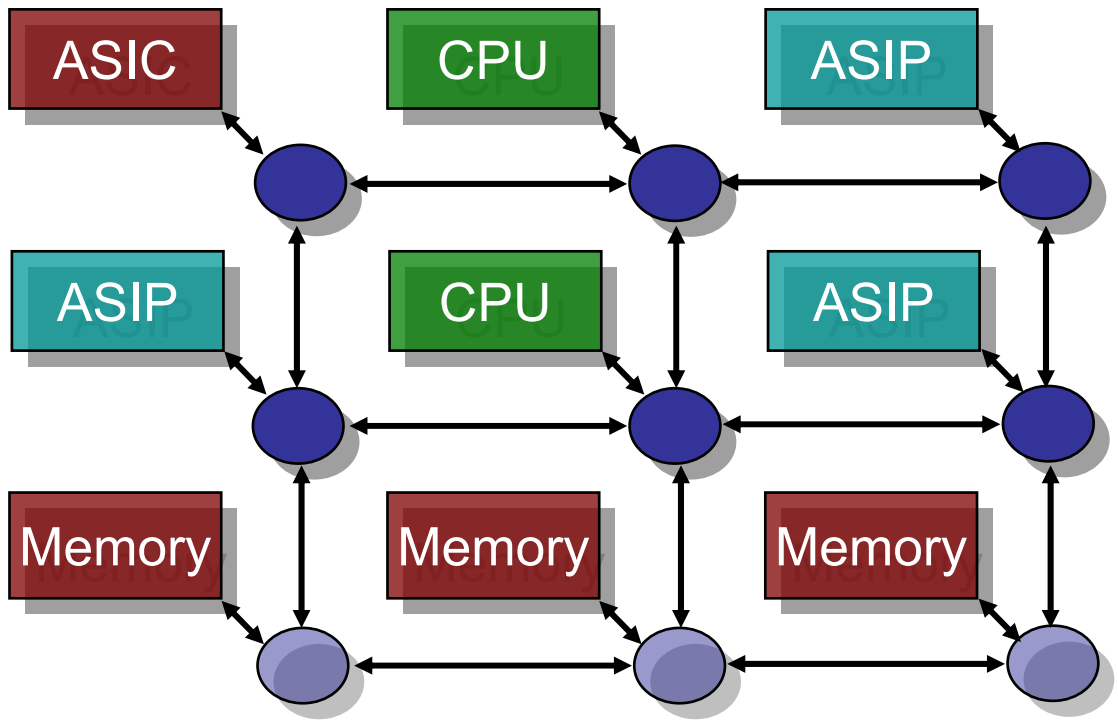
- Embedded SW design becoming predominant manpower factor in system design
- Cannot develop/maintain millions of code lines in assembly language
- Move to high-level programming languages



Why care about compilers?

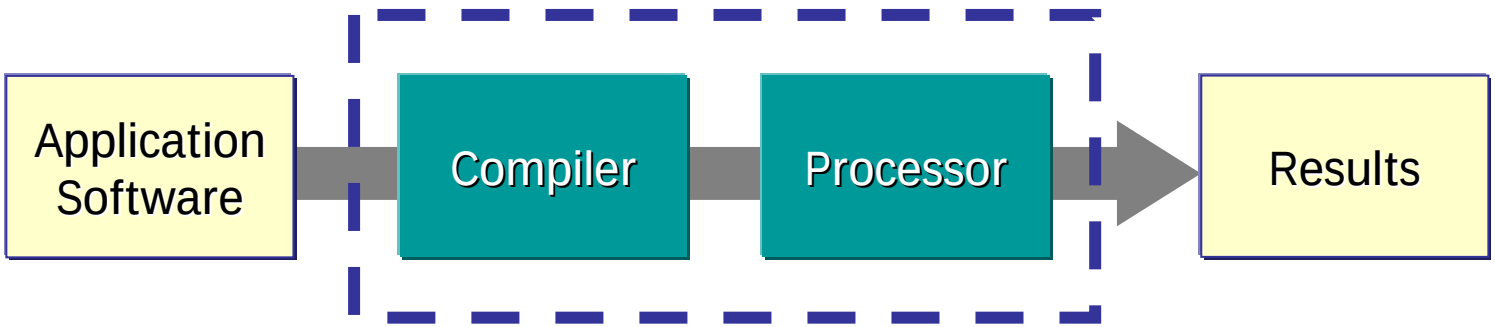
- Trend towards heterogeneous multiprocessor systems-on-chip (MPSoC)
- Customized application specific instruction set processors (ASIPs) are key MPSoC components
- How to achieve efficient compiler support for ASIPs?





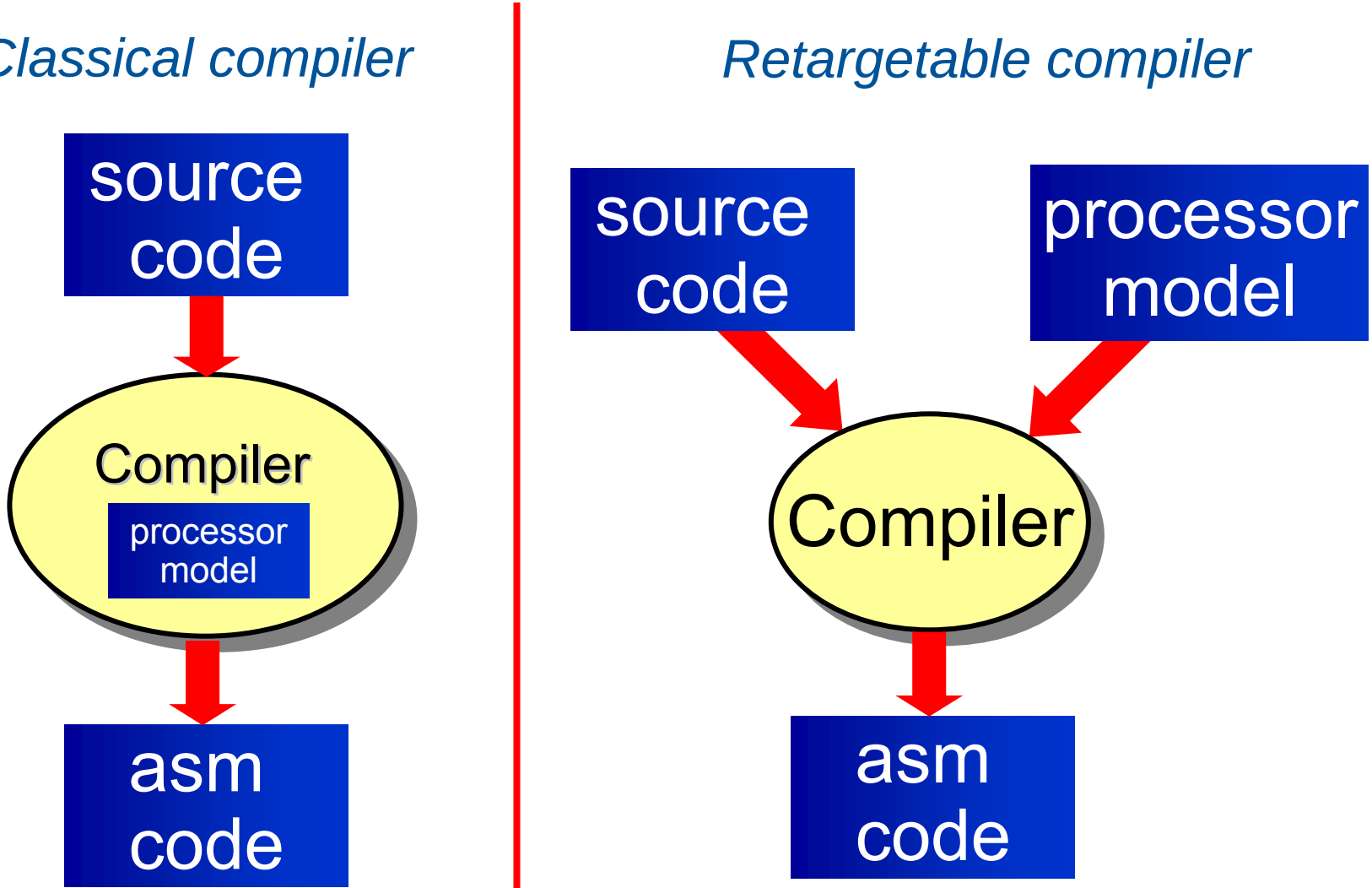
„Compiler/Architecture Co-Design“

- Efficient C-compilers cannot be designed for ARBITRARY architectures!



- Compiler and processor form a UNIT that needs to be optimized!
- “Compiler-friendliness“ needs to be taken into account during the architecture exploration!

Retargetable compilers



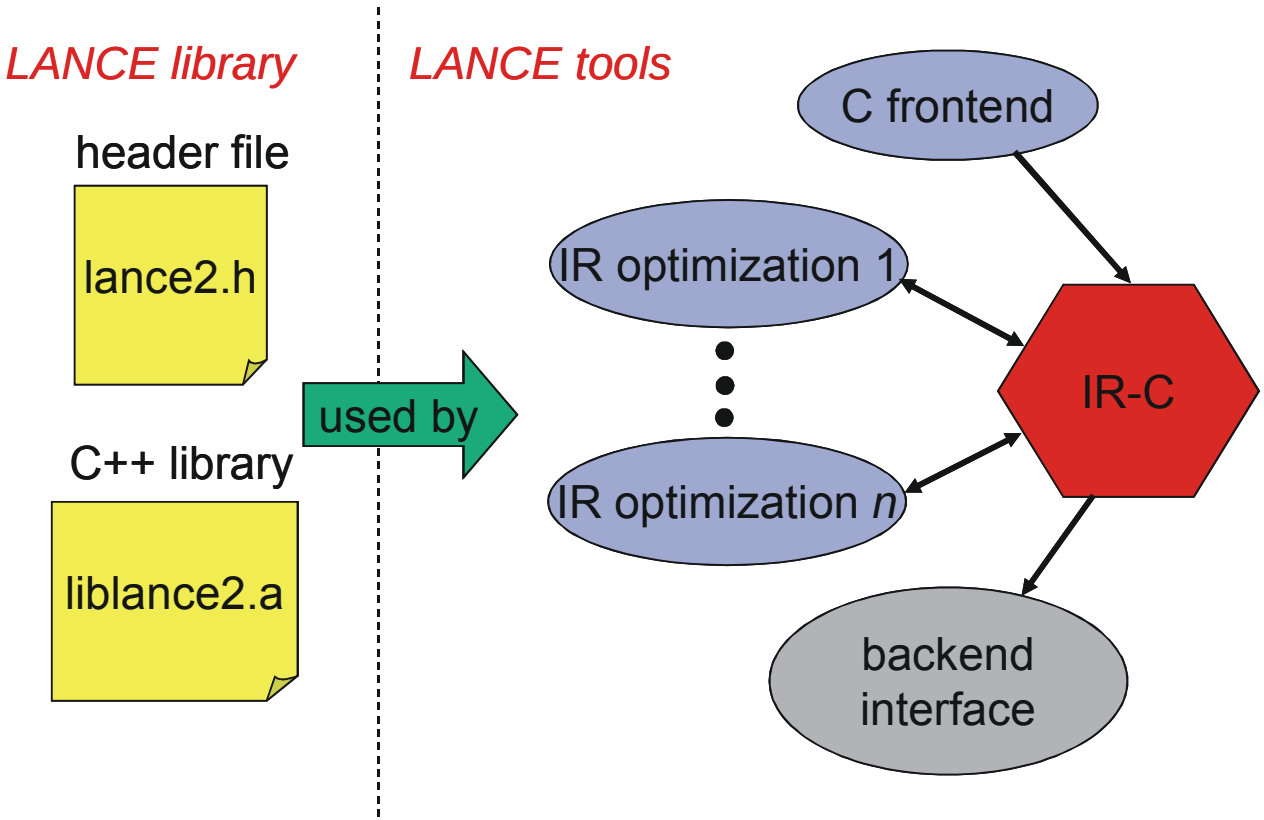
GNU C compiler (gcc)

- Probably the most widespread retargetable compiler
- Mostly used as a native Unix/Linux compiler, but may operate as a cross-compiler, too
- Support for C/C++, Java, and other languages
- Comes with comprehensive support software, e.g. runtime and standard libraries, debug support
- Portable to new architectures by means of machine description file and C support routines

> *"The main goal of GCC was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary."*

LANCE (Univ. of Dortmund/RWTH Aachen)

- Modular retargetable C compiler system
- www.lancecompiler.com



Executable C based IR in the LANCE compiler

C source code

```
void f()
{
  int i,A[10];
  i = A[2]++
      > 1 ?
      2 : 3;
}
```

```
int A[10];
char *t1,*t3;
int i,t2,t5,t6,t7,t8;
int *t4;
```

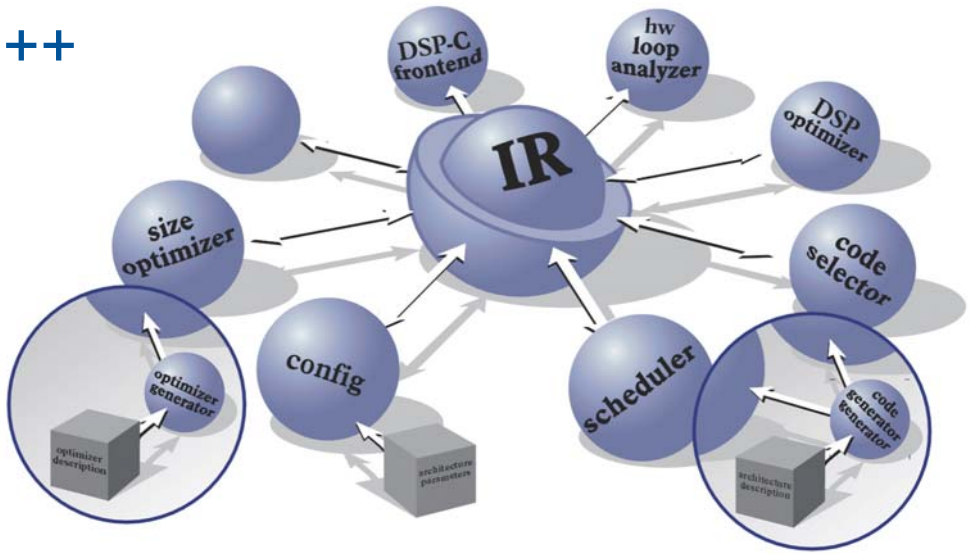
```
t3 = (char *)A;  // cast base to char*
t2 = 2 * 4;      // compute offset
t1 = t3 + t2;    // compute eff addr
t4 = (int *)t1;  // cast back to int*
t5 = *t4;        // load value from memory
```

```
t6 = t5 + 1;     // increment
*t4 = t6;        // store back into A[2]
```

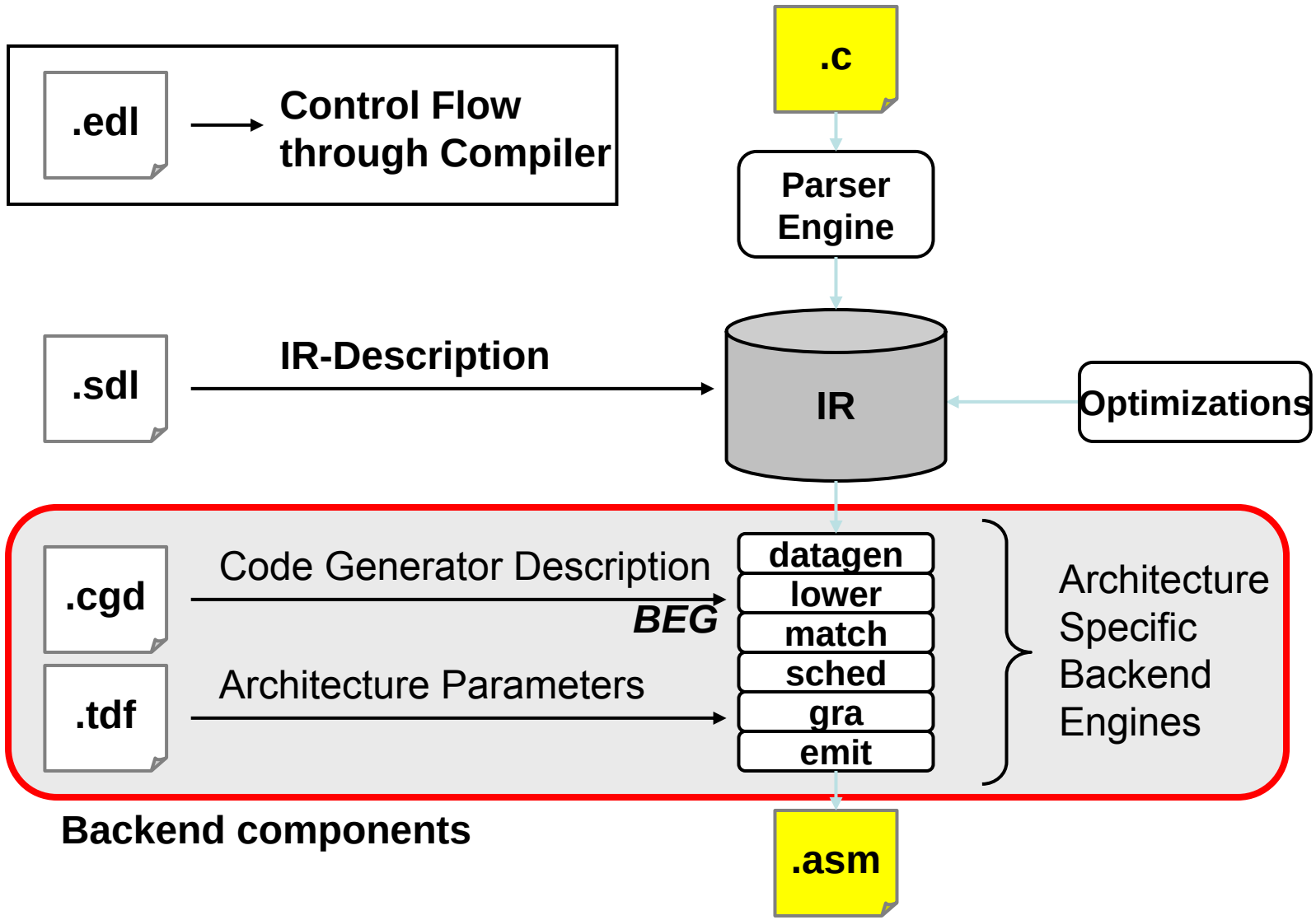
```
    t7 = t5 > 1;       // compare
    if (t7) goto L1;   // jump if >
    t8 = 3;            // load 3 if <=
    goto L2;           // goto join point
L1: t8 = 2;            // load 2 if >
L2: i = t8;            // move result into i
```

CoSy compiler system (ACE)

- Universal retargetable C/C++ compiler
- Extensible intermediate representation (IR)
- Modular compiler organization
- Generator (BEG) for code selector, register allocator, scheduler



ACE CoSy system structure



LISATek C compiler generation

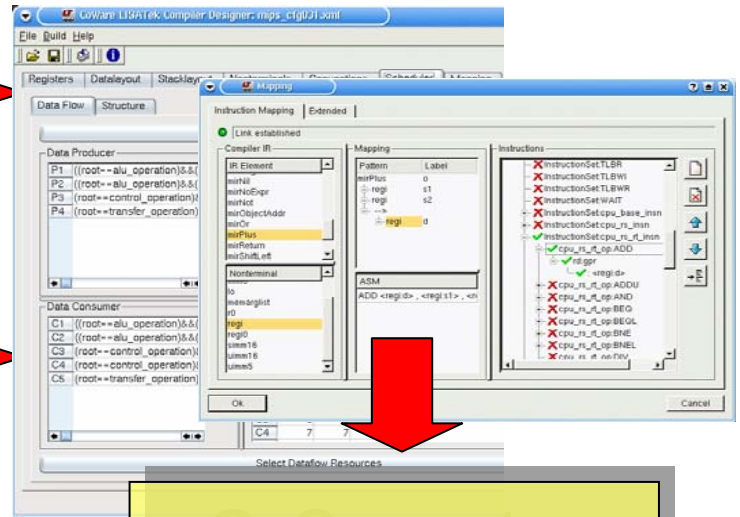
LISA processor model

```
SYNTAX      {
      "ADD" dst, src1, src2
}
CODING      {
      0b0010 dst src1 src2
}
BEHAVIOR    {
      ALU_read (src1, src2);
      ALU_add ();
      Update_flags ();
      writeback (dst);
}
SEMANTICS   {
      src1 + src2 → dst;
}
...
```

Autom. analyses

Manual refinement

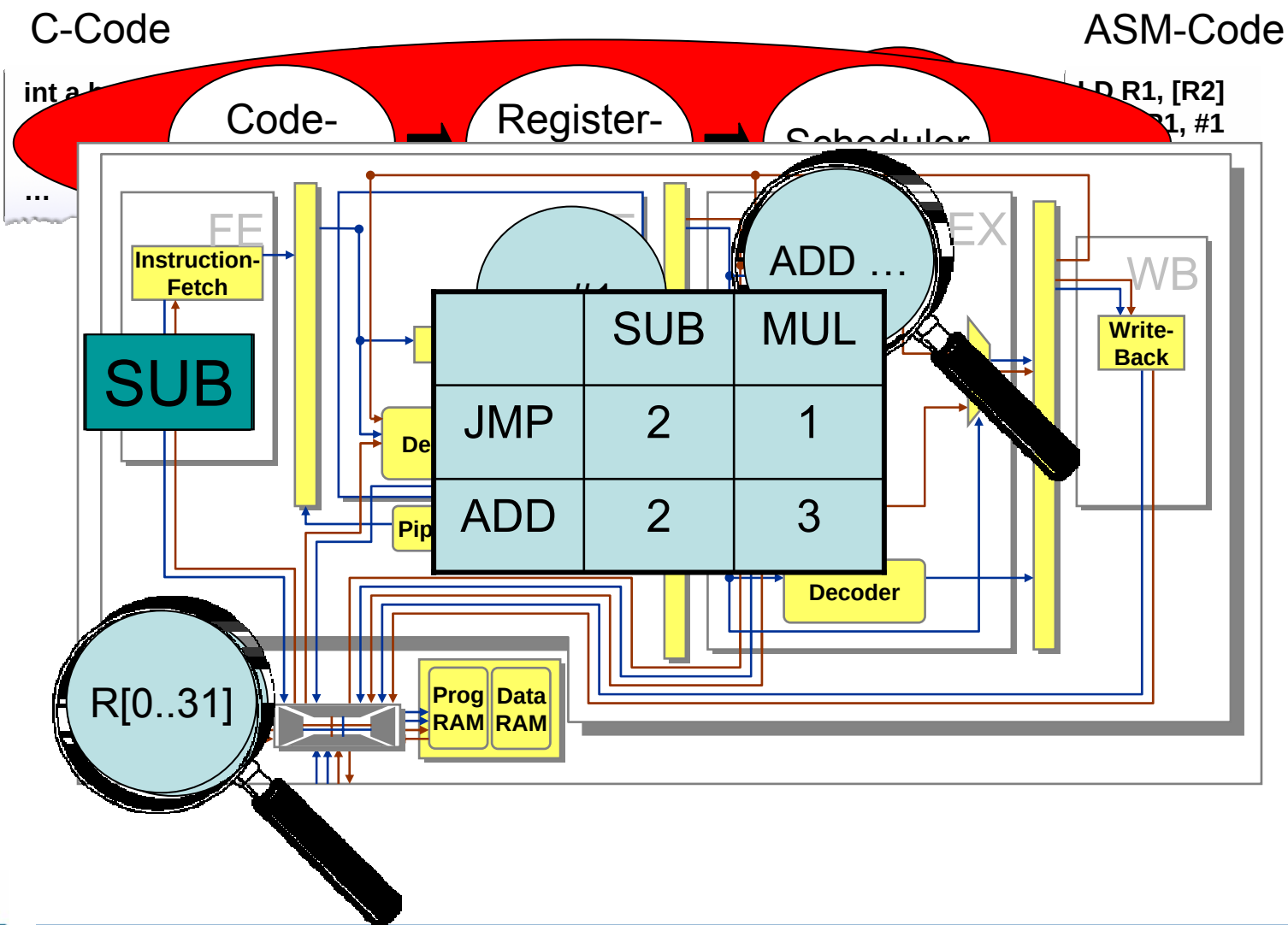
GUI



CoSy system

C Compiler

LISATek compiler generation

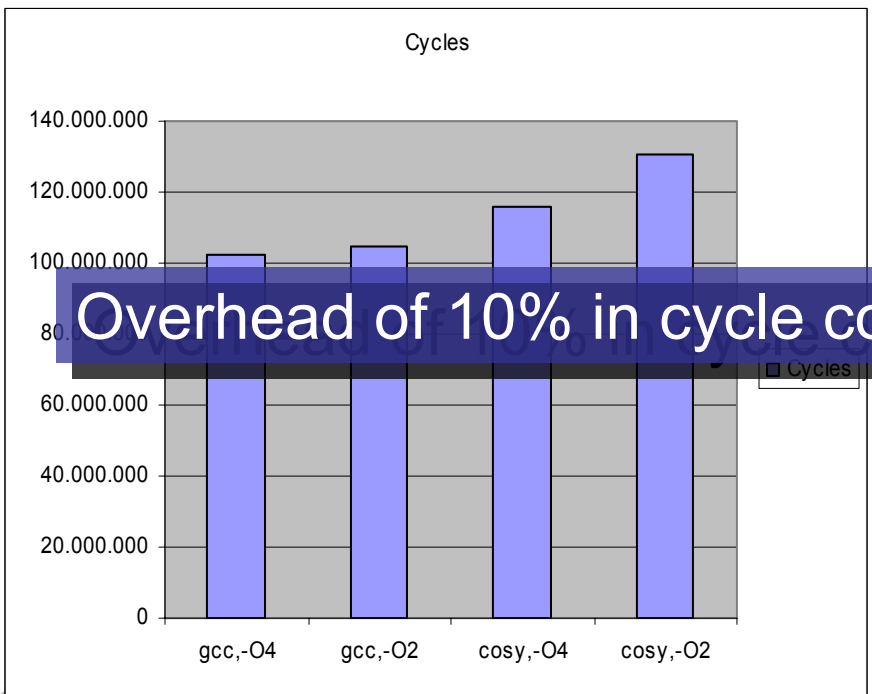


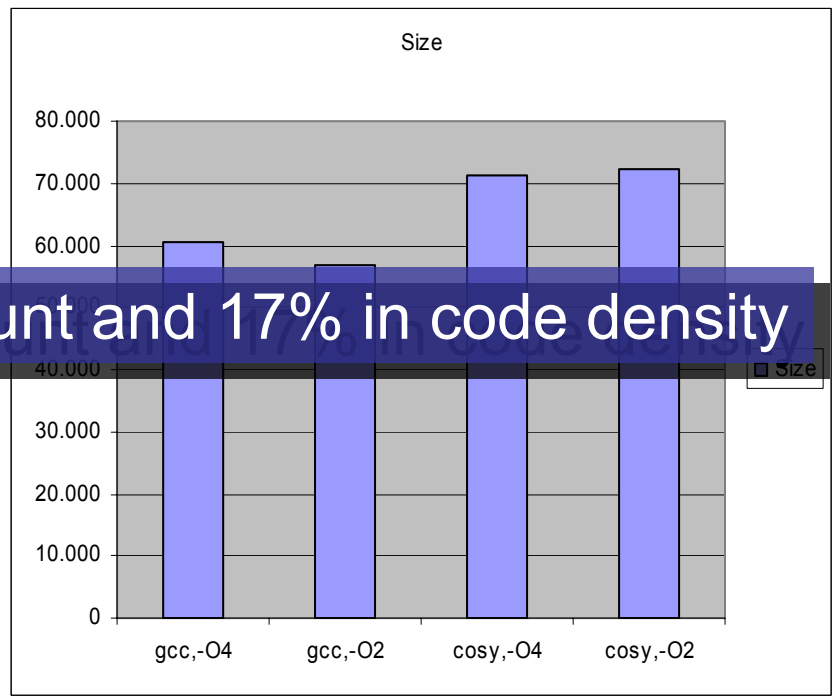
Compiled code quality: MIPS example

- LISATek generated C-Compiler
- Out-of-the-box C-Compiler
- No manual optimizations
- Development time of model approx. 2 weeks

gcc C-Compiler

- gcc with MIPS32 4kc backend
- Used by most MIPS users
- Large group of developers, several man-years of optimization



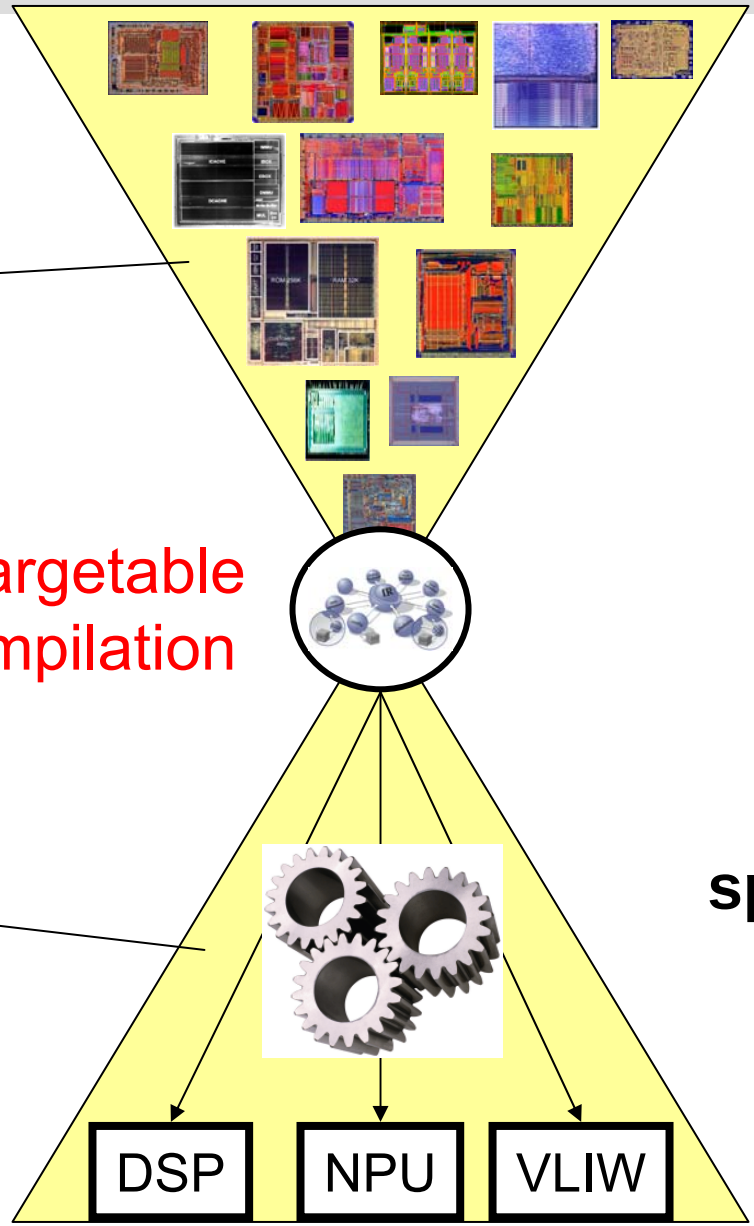


Overhead of 10% in cycle count and 17% in code density

Demands on code quality

- Compilers for embedded processors have to generate extremely efficient code
  - Code size:
    - » system-on-chip
    - » on-chip RAM/ROM
  - Performance:
    - » real-time constraints
  - Power/energy consumption:
    - » heat dissipation
    - » battery lifetime

Compiler flexibility/code quality trade-off

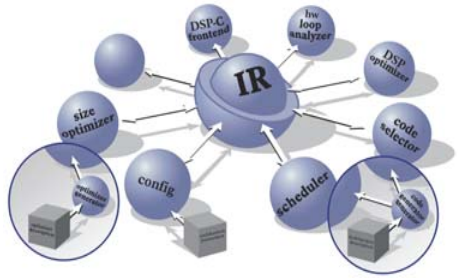


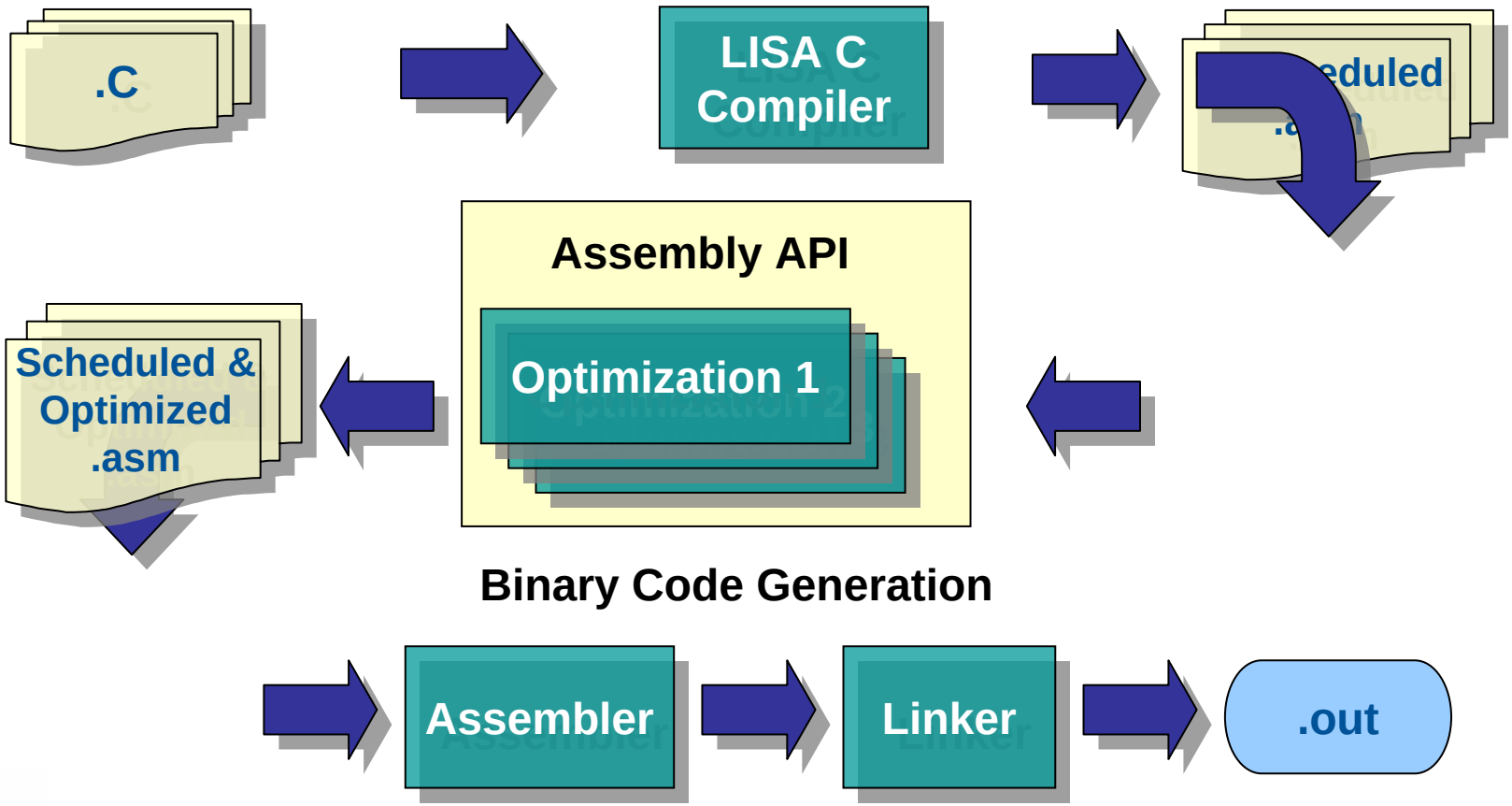
unification

specialization

Adding processor-specific code optimizations

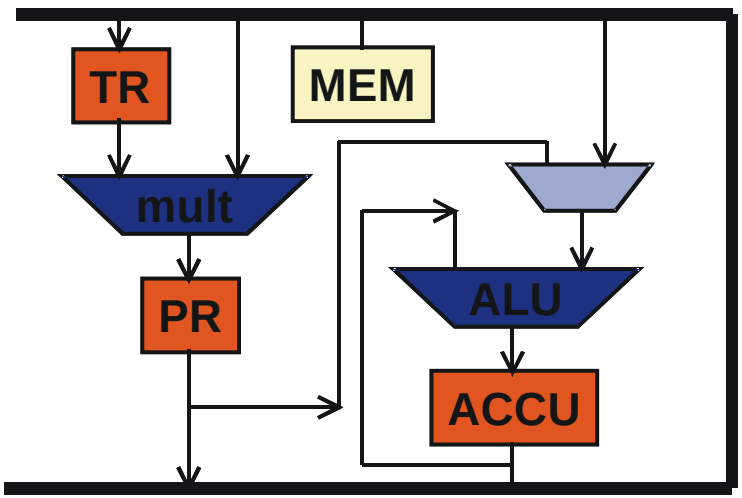
- High-level (compiler IR)
  - Enabled by CoSy´s engine concept
- Low-level (ASM):



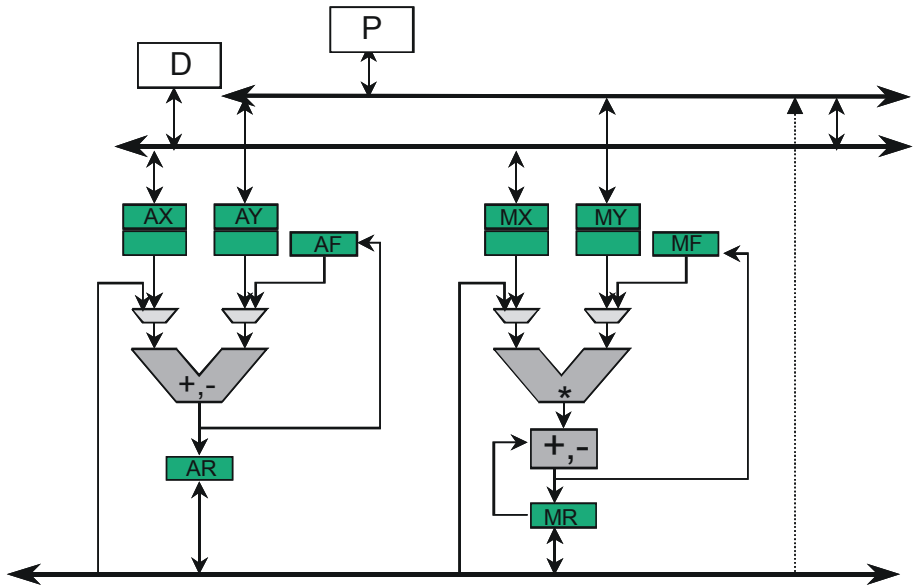


Embedded processors are not compiler-friendly

- Designed for efficiency
- E.g. fixed-point DSP data paths:



TI C25

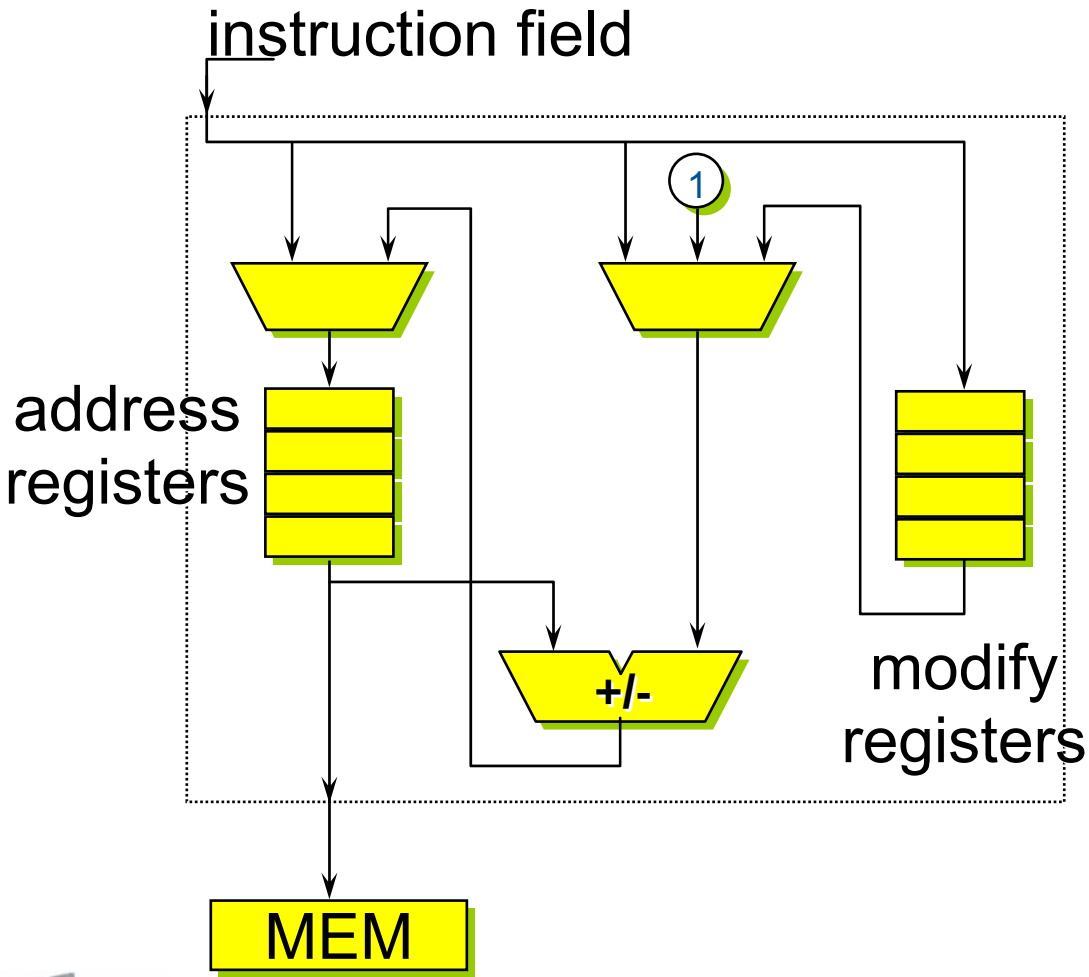


ADSP210x

- Special purpose registers, constrained parallelism, ...
- Challenge for compilers that usually prefer orthogonal „compiler-friendly“ architectures

DSP address code optimization

- Based on address generation unit (AGU) support
- Address arithmetic in parallel to central data path



Support for auto-increment and auto-modify

Examples:

- TI C2x/C5x
- Motorola 56000
- ADSP-210x
- AMS Gepard core

DSP address code optimization

variable set: { a, b, c, d }

access sequence: b, d, a, c, d, a, c, b, a, d, a, c, d

alphabetic layout

0	a
1	b
2	c
3	d

AR = 1
AR += 2
AR -= 3
AR += 2
AR ++
AR -= 3
AR += 2
AR - -
AR - -
AR += 3
AR -= 3
AR += 2
AR ++

cost: 9

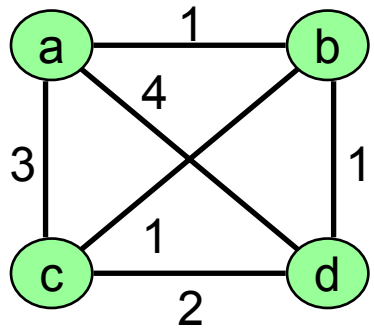
optimized layout

0	c
1	a
2	d
3	b

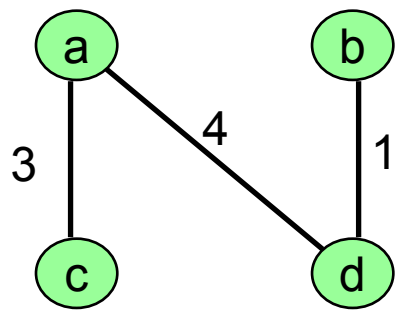
AR = 3
AR - -
AR - -
AR - -
AR += 2
AR - -
AR - -
AR += 3
AR -= 2
AR ++
AR - -
AR - -
AR += 2

cost: 5

access graph:

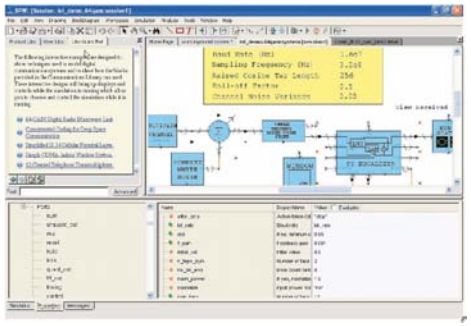


max. Hamiltonian path:



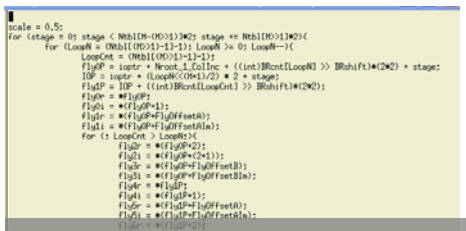
www.address-code-optimization.org

Source-level code optimization



Algorithm design

– better performance
– lower code size
– highly reusable



C code generation

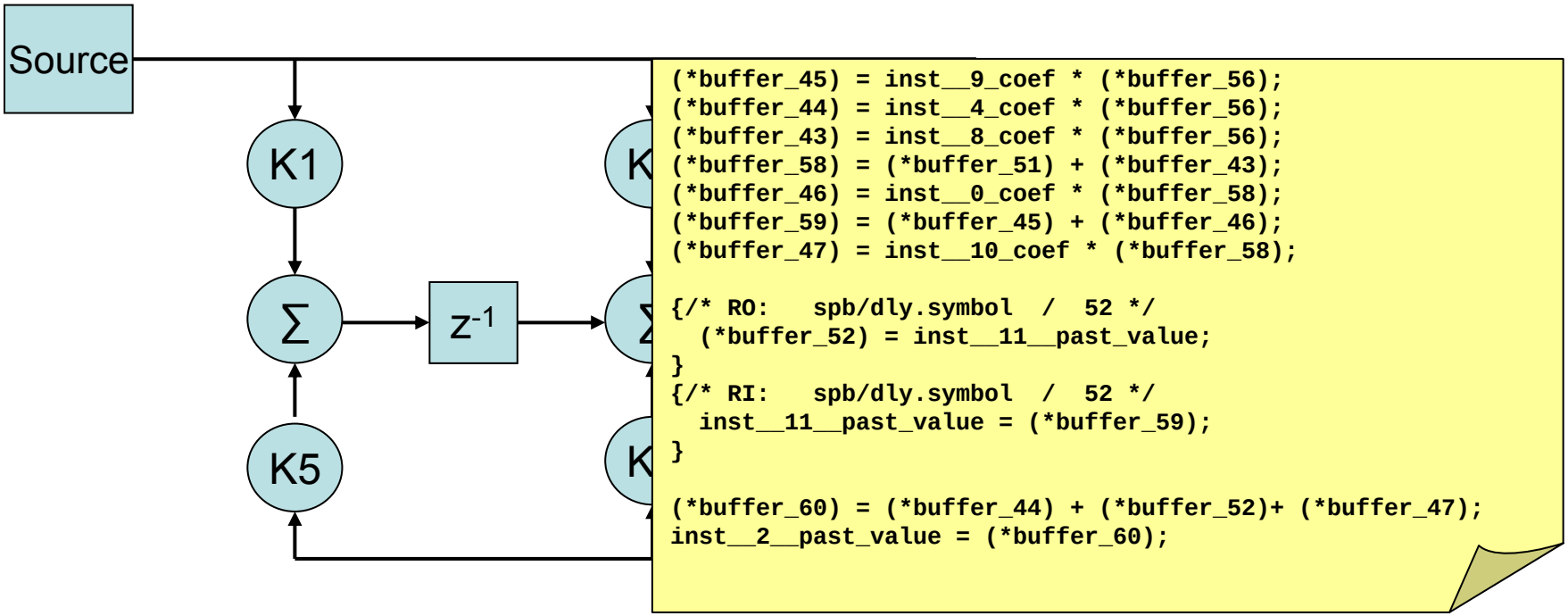
Poor code quality!

„Software washing“

„Software washing machine“

Compilation to assembly

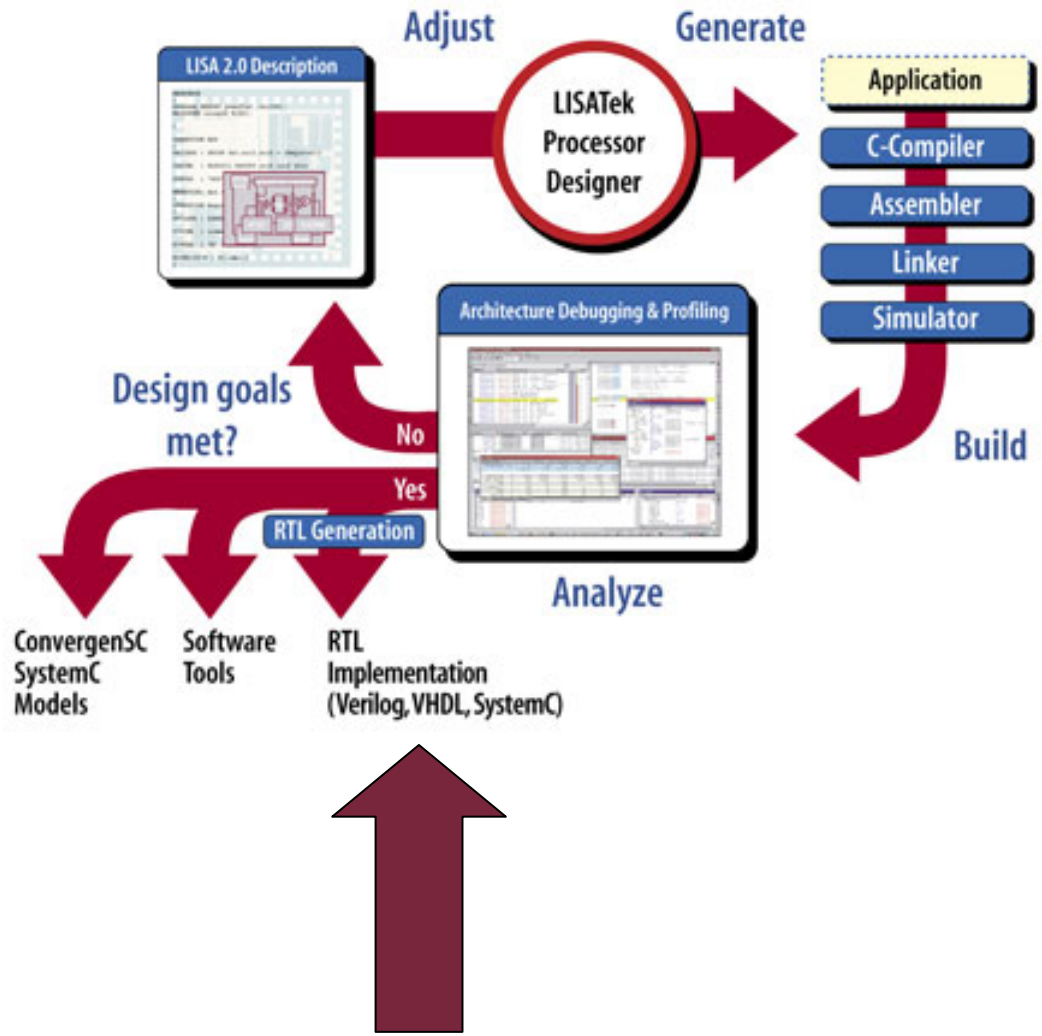
CoWare SPW example



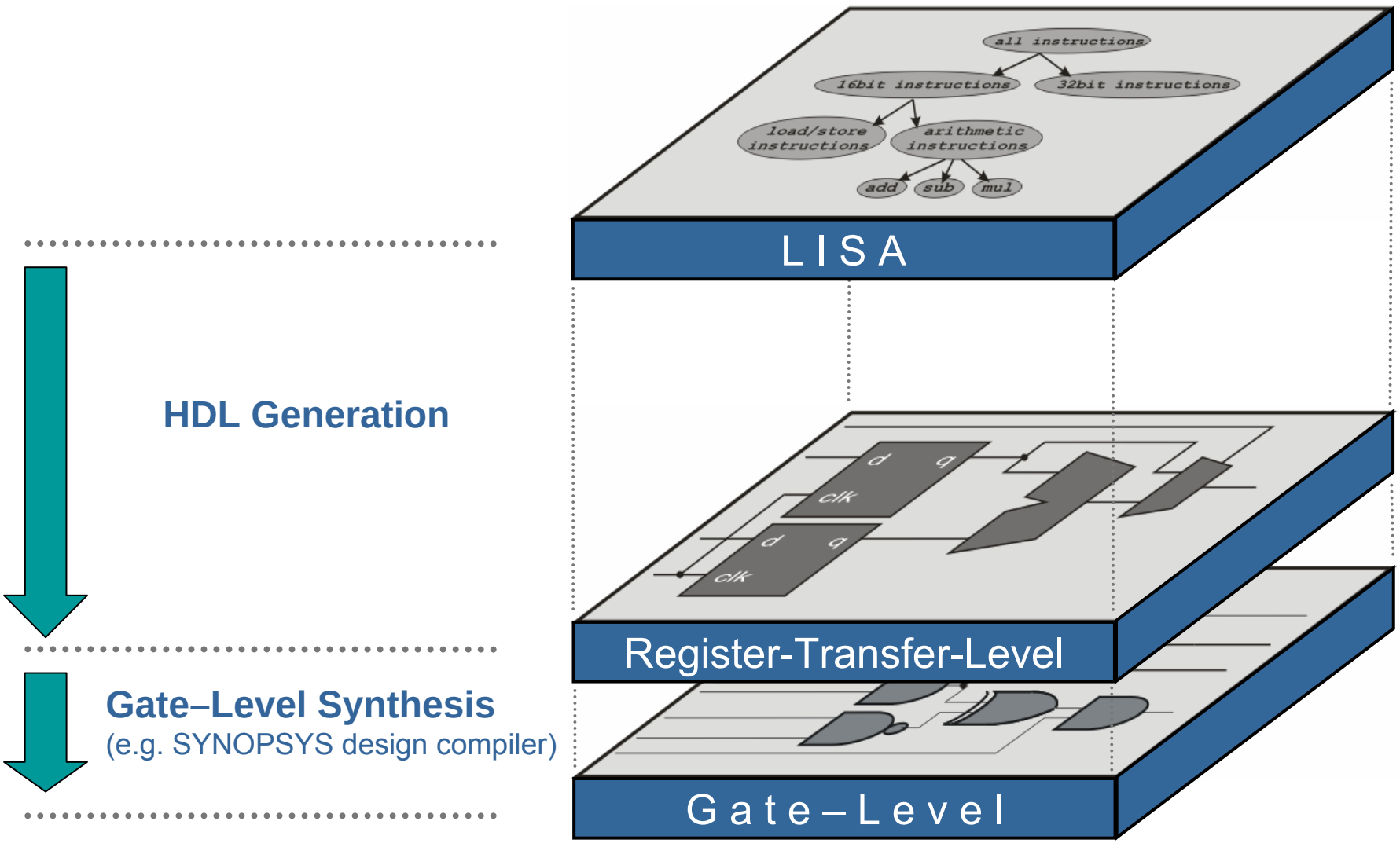
- Very conservative, block-oriented code generation scheme
- Even simple code transformations have great impact, e.g.
  - Variable localization
  - If-statement merging
  - Optimized data type selection

4. ASIP architecture design

ASIP implementation after exploration

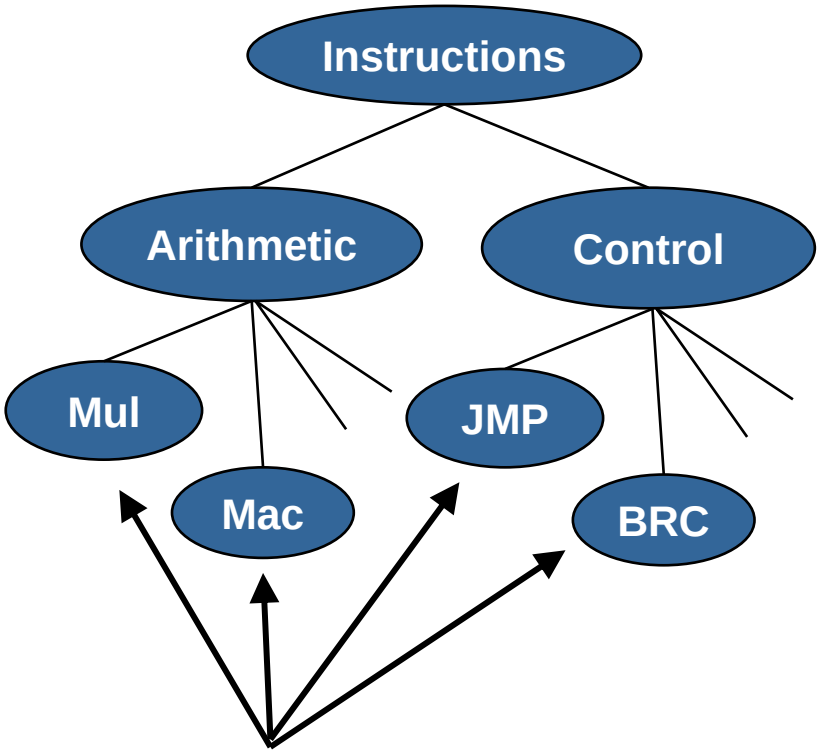


Unified Description Layer

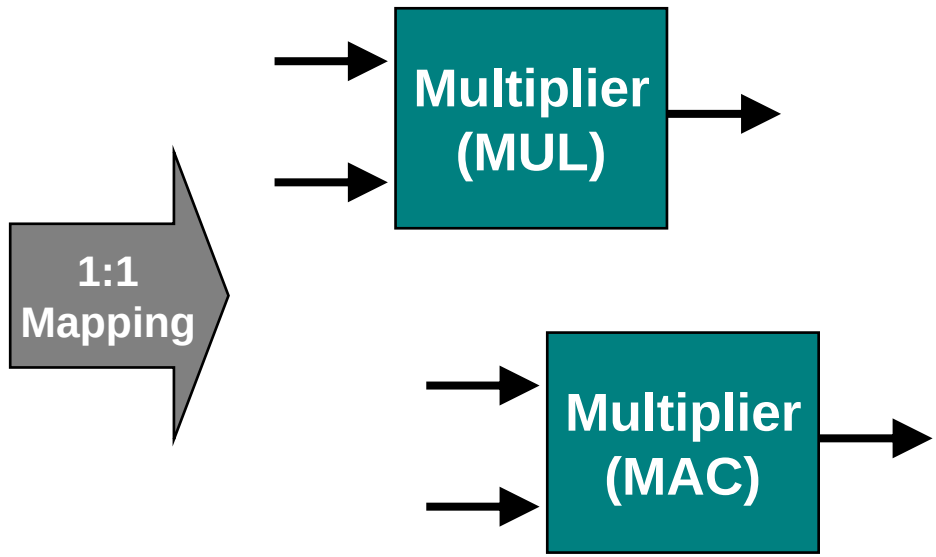


Challenges in Automated ASIP Implementation

ADL:



HDL:



Independent description of instruction behavior:

+ Efficient Design Space Exploration

Independent mapping to hardware blocks:

- Insufficient architectural efficiency by 1:1 mapping

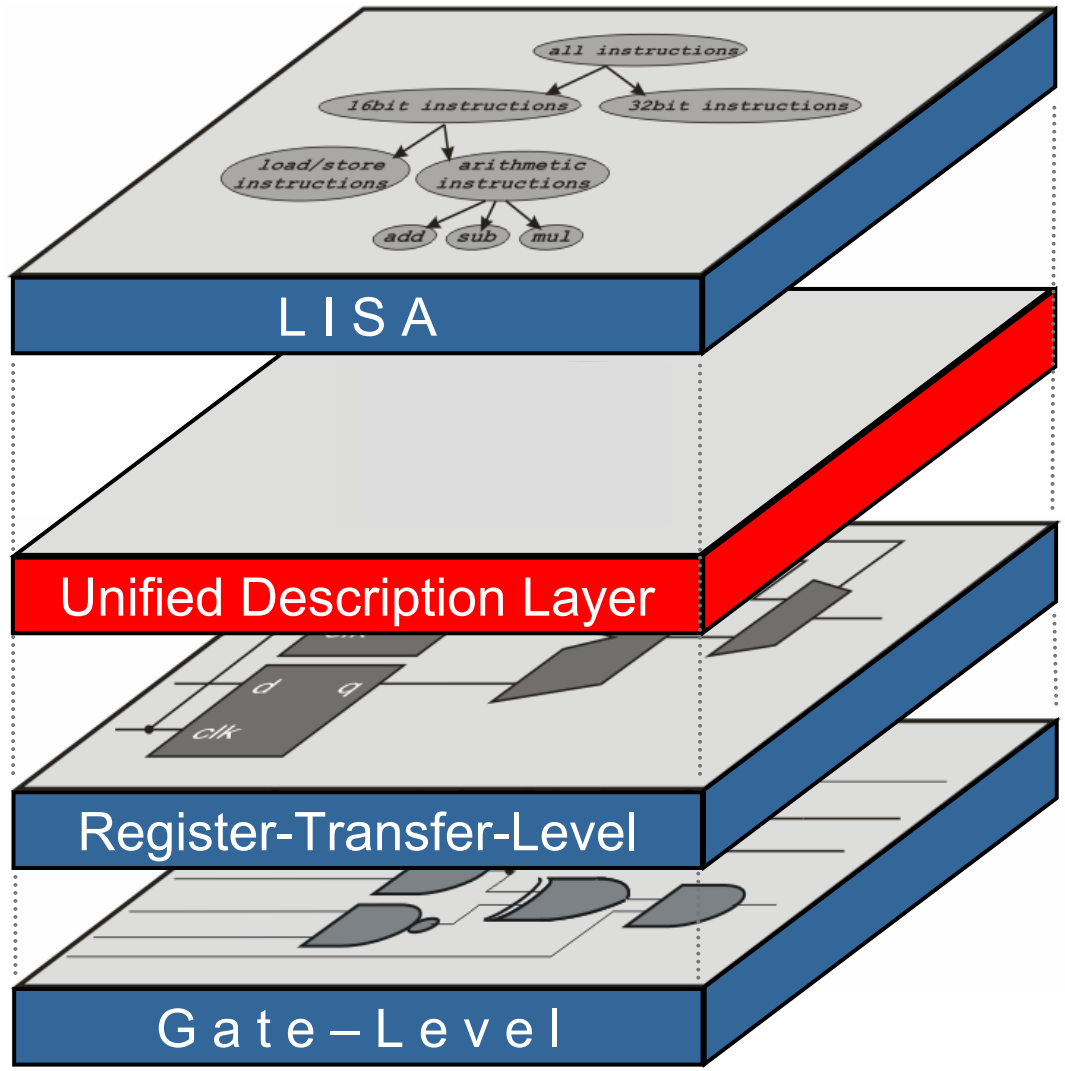
Unified Description Layer

Structure & Mapping
(incl. JTAG/DEBUG)

Optimizations

Backend (VHDL, Verilog, SystemC)

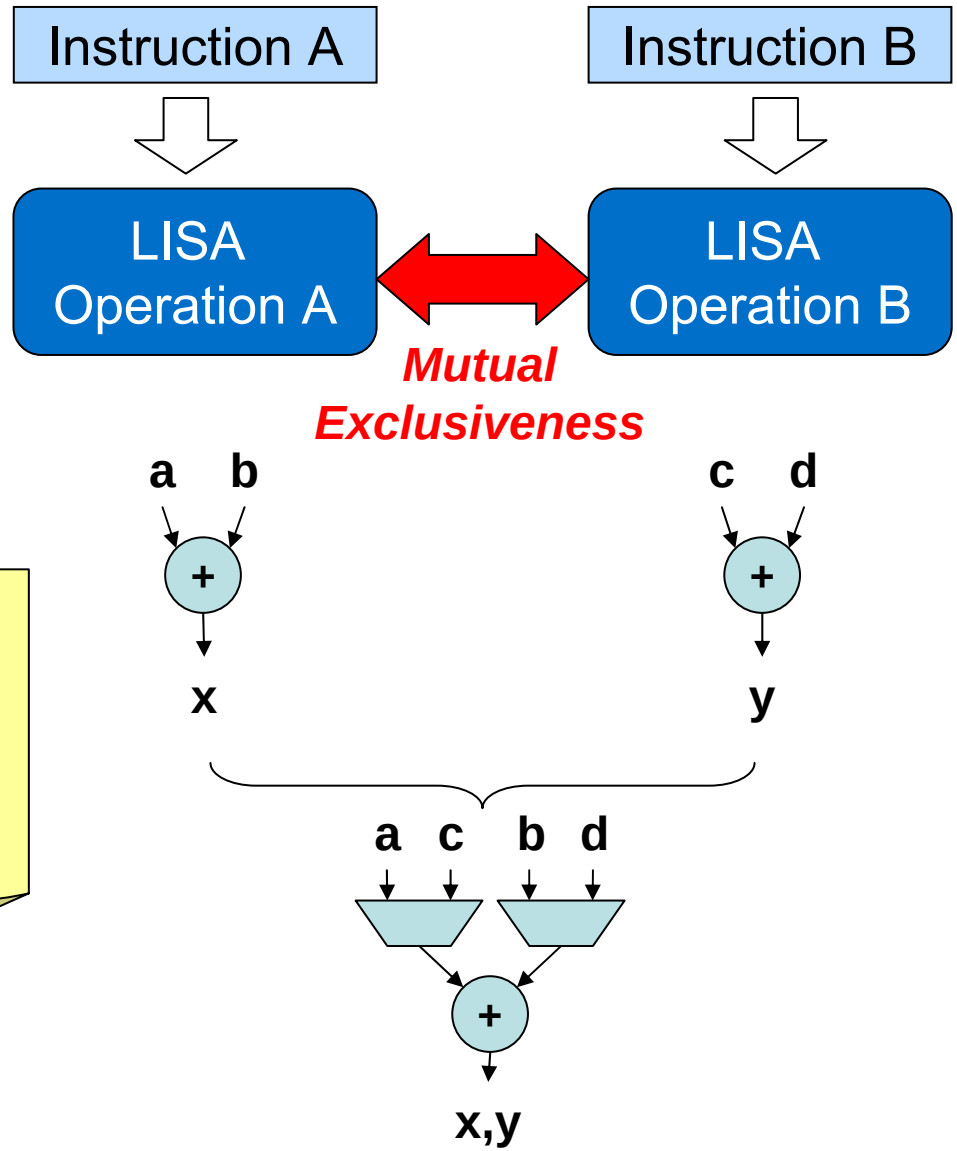
Gate–Level Synthesis
(e.g. SYNOPSYS design compiler)



Optimization strategies

LISA: separate descriptions for separate instructions

Goal: share hardware for separate instructions



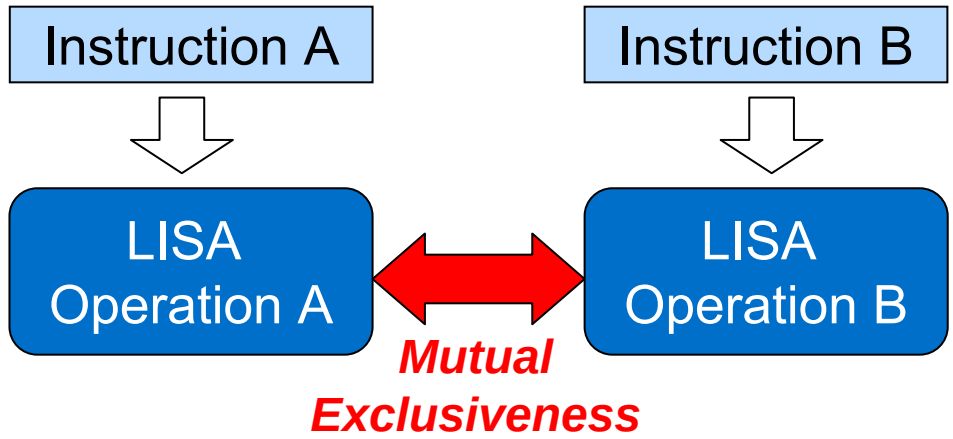
Possible Optimizations

- ALU Sharing

Optimization strategies

LISA: separate descriptions for separate instructions

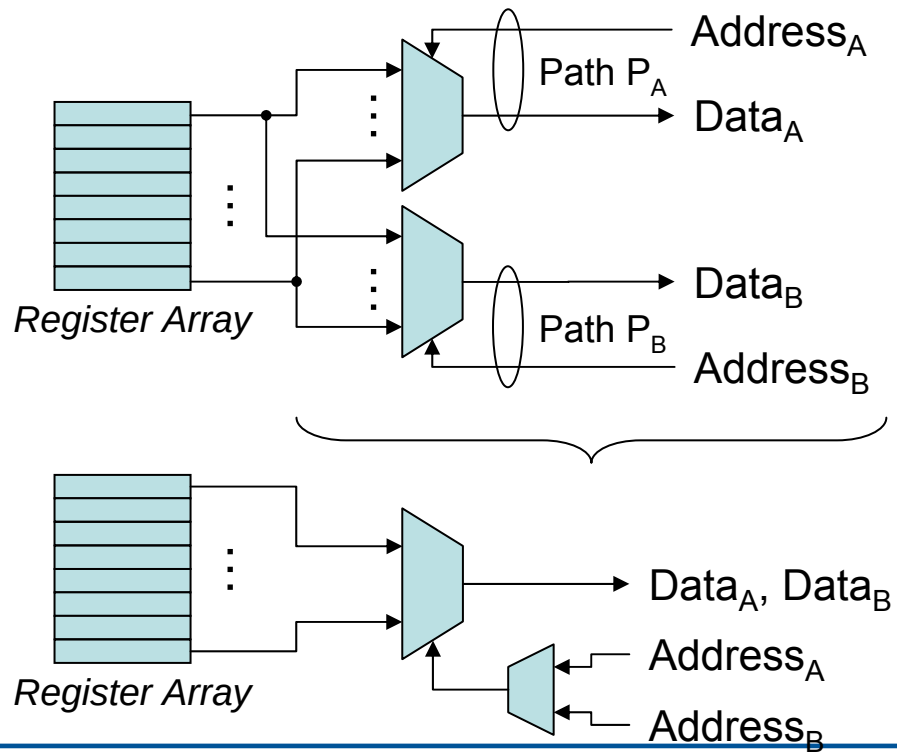
Goal: same hardware for separate instructions



Possible Optimizations

- ALU Sharing
- Path Sharing
- ...

} *Resource Sharing*



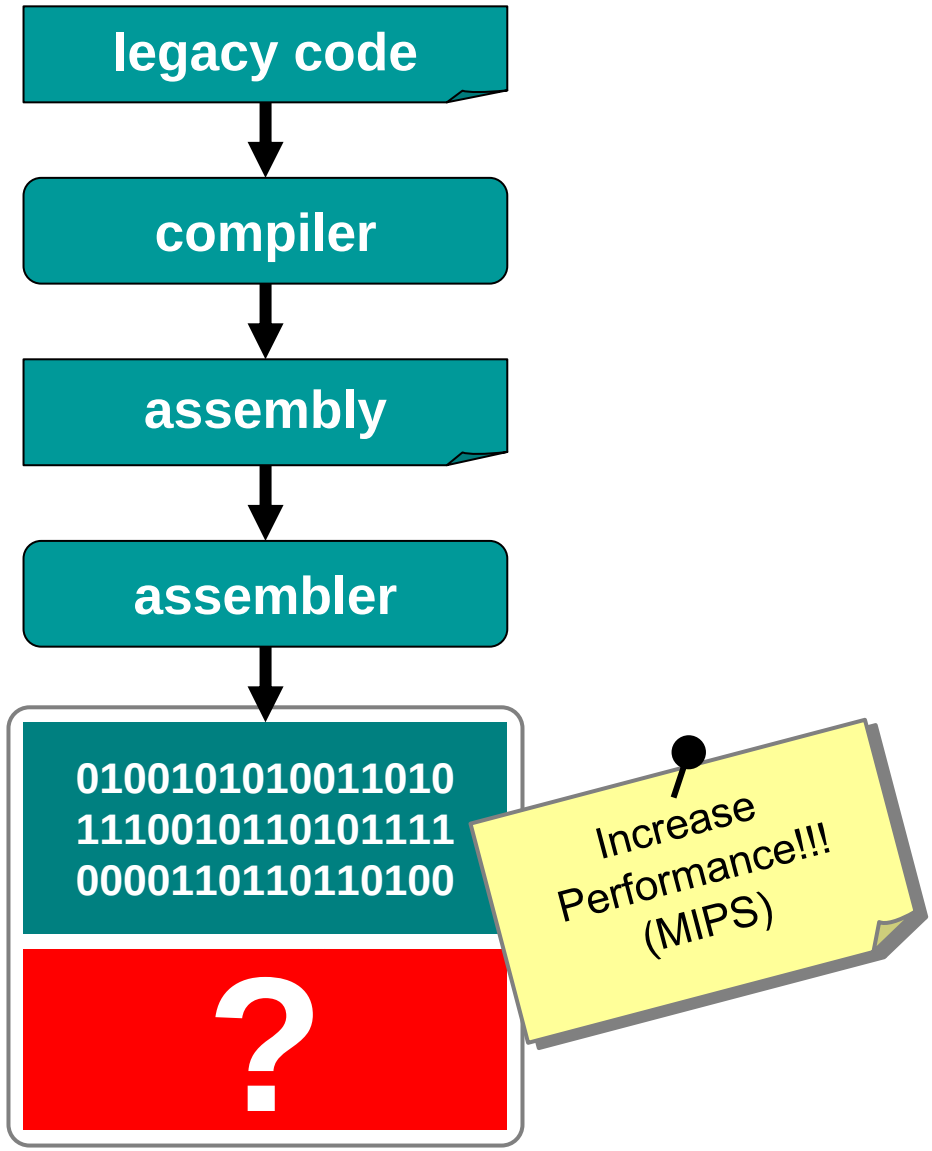
5. Case study

Project Goals:

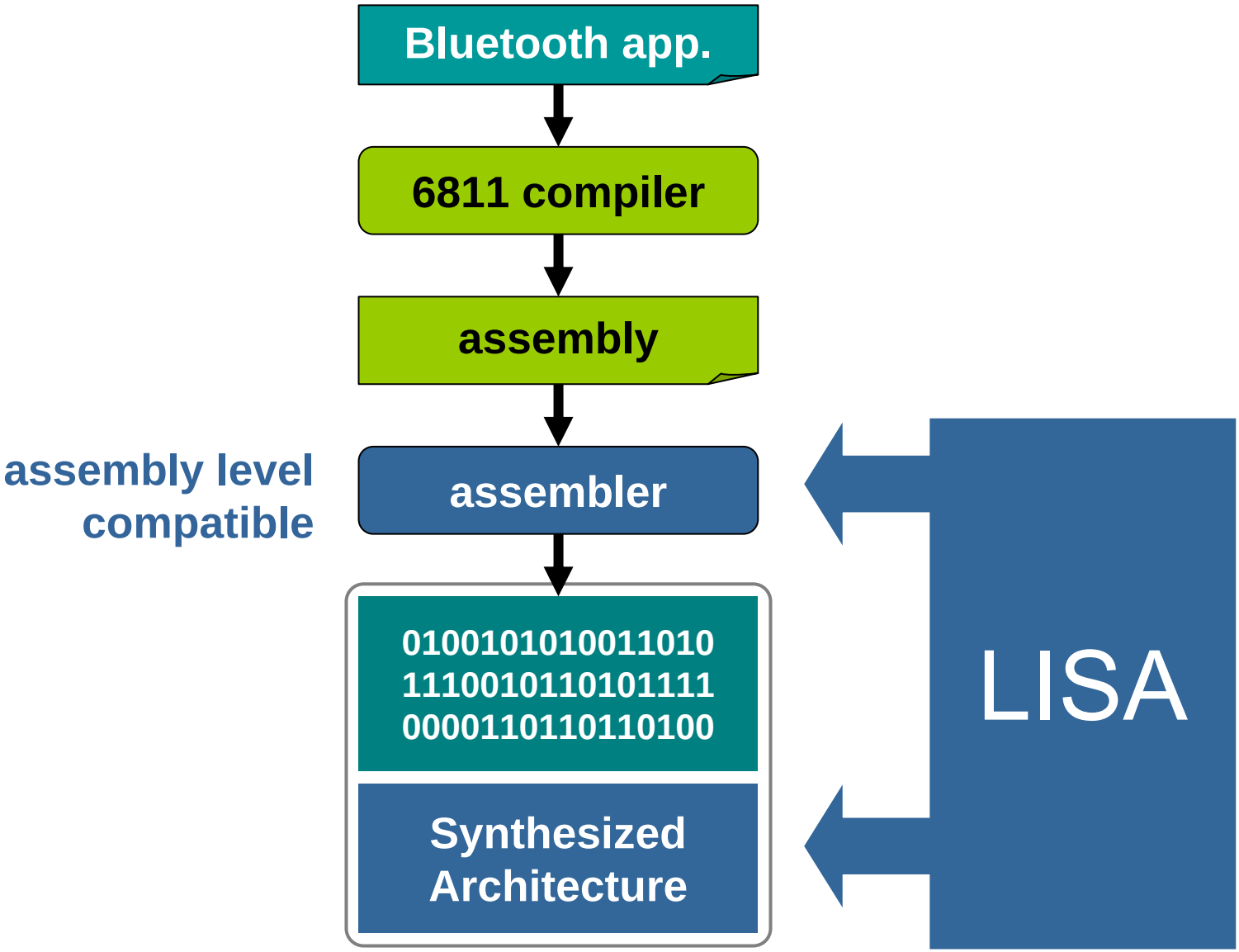
- Performance (MIPS) must be increased
- Compatibility on the assembly level
  for reuse of legacy code
  (Integration into existing tool flow)
- Royalty free design

➔ compatible architecture developed with LISA
  using RTL processor synthesis

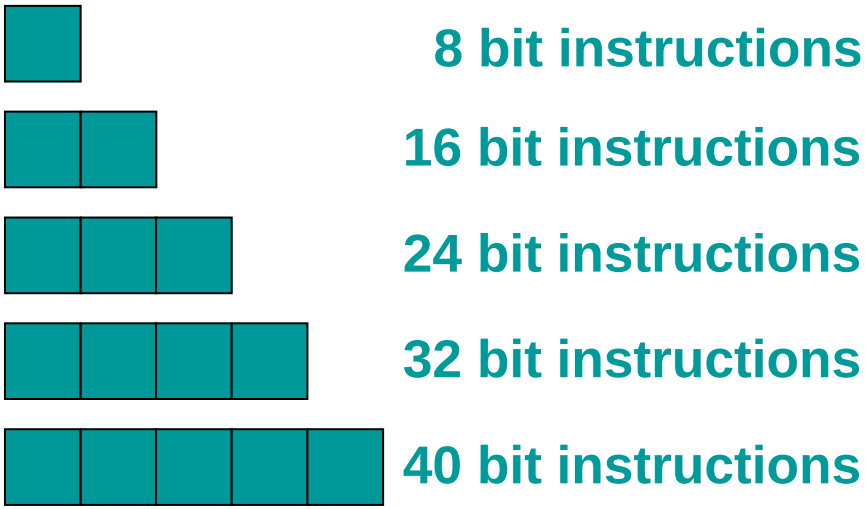
Motorola 6811



Motorola 6811



original 6811 Processor



8 bit instructions

16 bit instructions

24 bit instructions

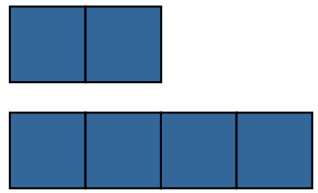
32 bit instructions

40 bit instructions

Instruction is fetched by 8 bit blocks:

➔ up to 5 cycles for fetching!

LISA 6811 Processor



16 bit instructions

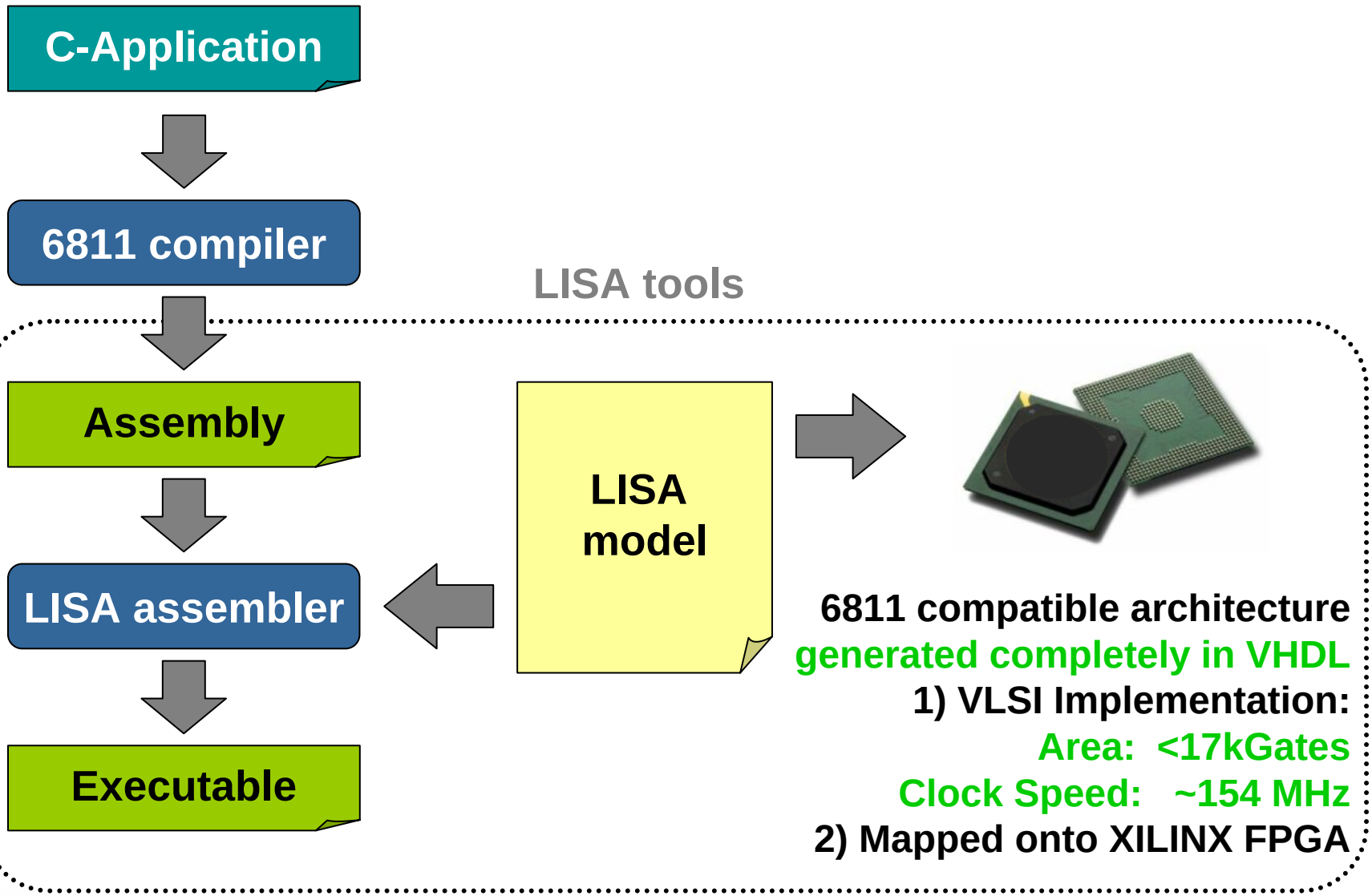
32 bit instructions

16 bit are fetched simultaneously:

➔ max 2 cycles for fetching!

\+ pipelined architecture
\+ possibility for special instructions

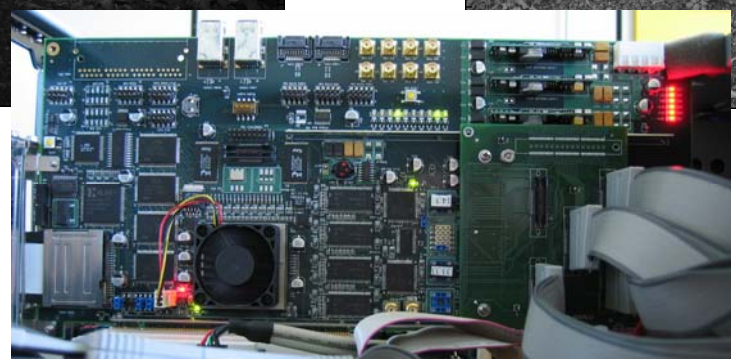
Tools Flow and RTL Processor Synthesis



Retinex ASIP (digital image enhancement)

IN

OUT



Virtex II FPGA board

ASIC (0.18 µm):

- **102 kGates, 93 MHz**

ASIP (0.13 µm):

- **124 kGates, 154 MHz**
- **SW programmable!**

6. Advanced research topics

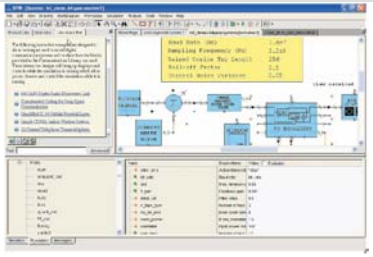
Generalized ASIP architecture design flow

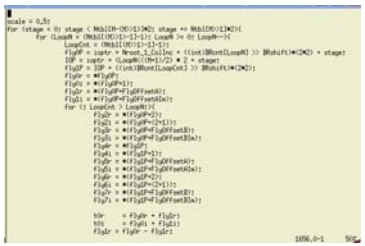
Algorithm design (Matlab, SPW, ...)

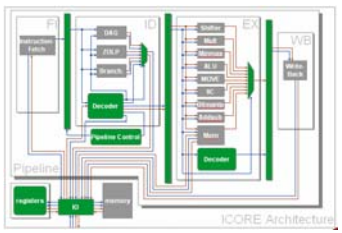
C code generation or implementation

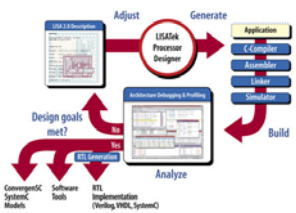
initial architecture

architecture optimization

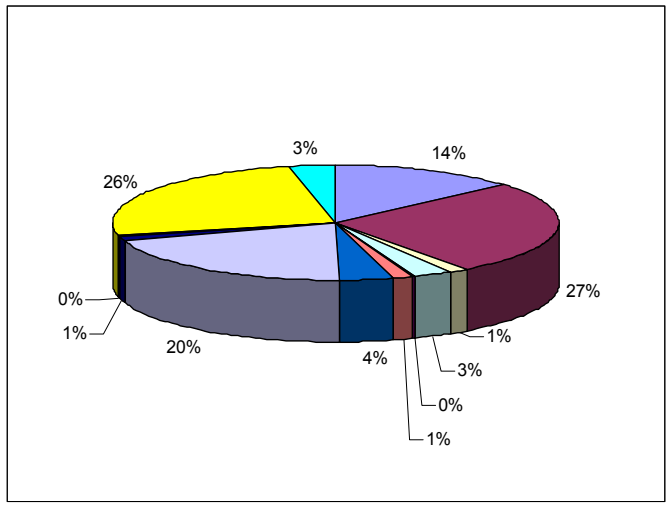








Profiling



A closer look at profilers: ASM level

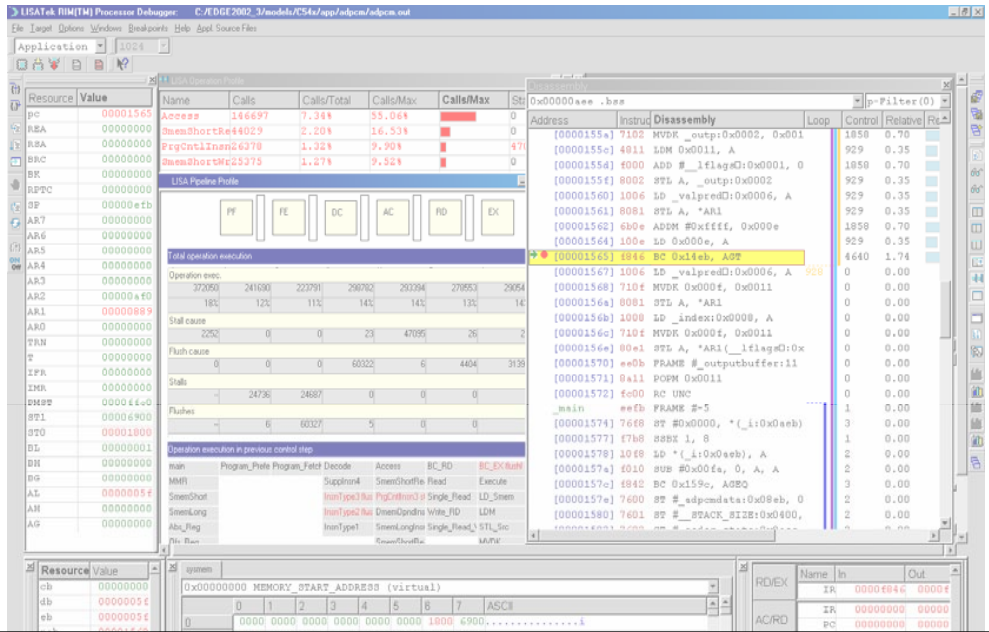
Algorithm design (Matlab, SPW, ...)

C code generation or implementation

initial architecture

architecture optimization

ASM level



– requires full architecture description
– slow (~1000x native C)

A closer look at profilers: source level

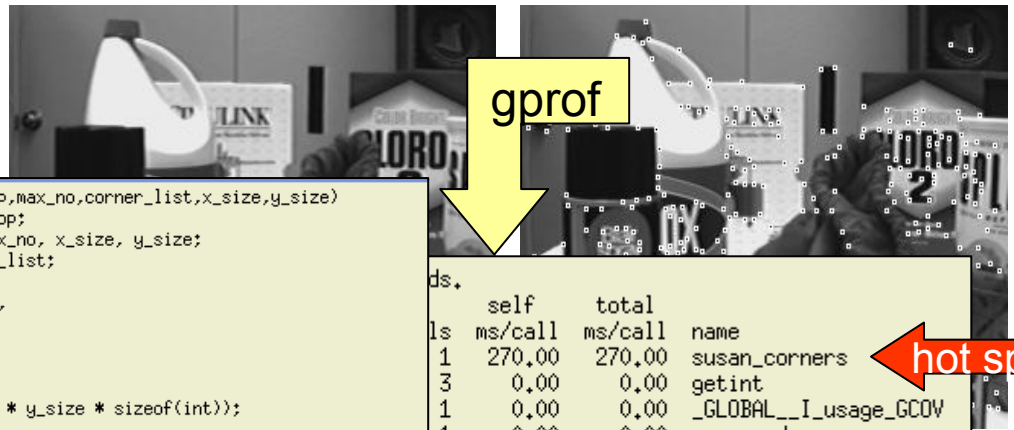
Algorithm design (Matlab, SPW, ...)

C code generation or implementation

initial architecture

architecture optimization

sample app: image corner detection



gprof

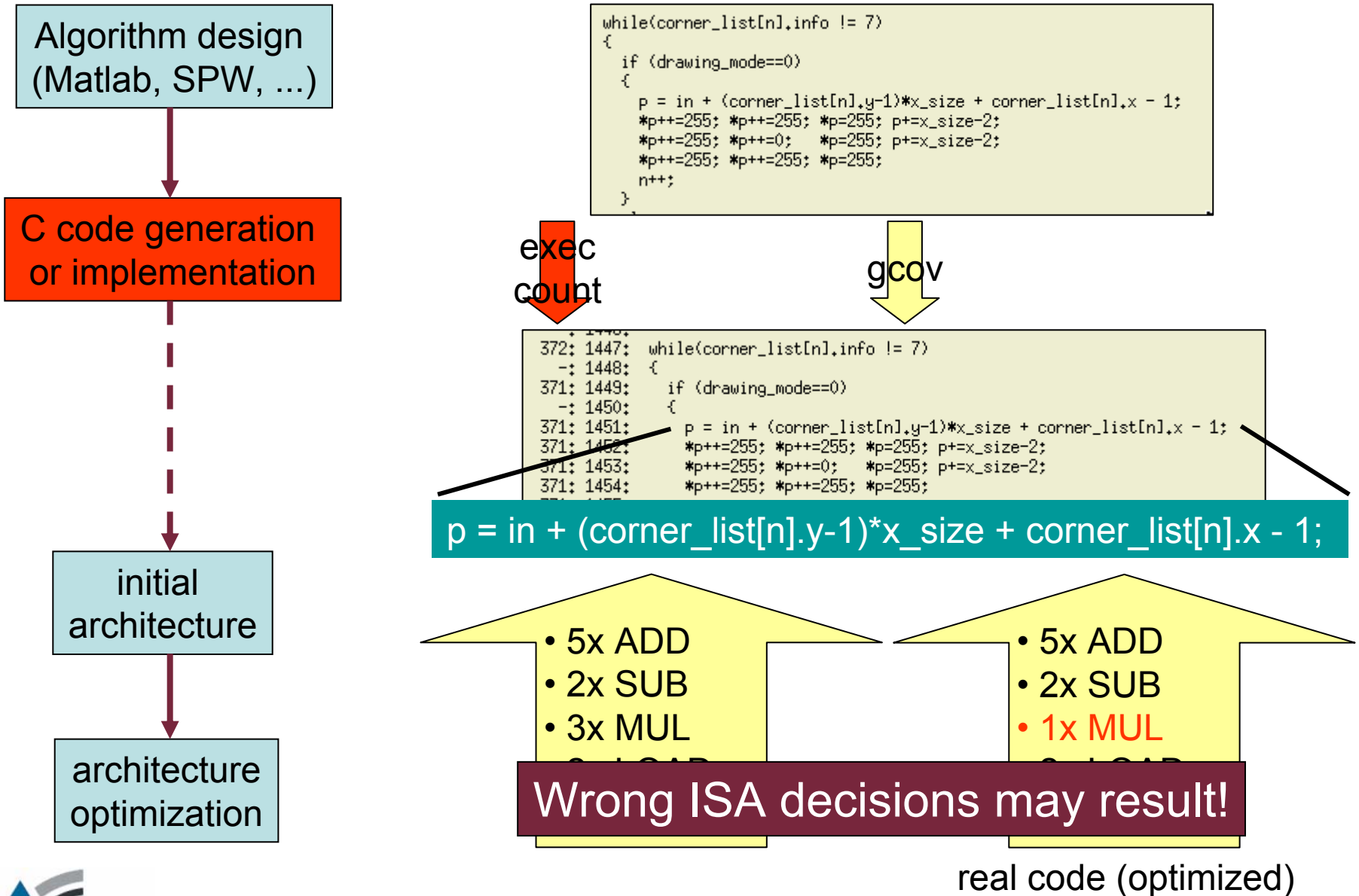
```
susan_corners(in,r,bp,max_no,corner_list,x_size,y_size)
  uchar       *in, *bp;
  int         *r, max_no, x_size, y_size;
  CORNER_LIST corner_list;
{
int   n,x,y,sq,xx,yy,
      i,j,*cgx,*cgy;
float divide;
uchar c,*p,*cp;

  memset (r,0,x_size * y_size * sizeof(int));

  cgx=(int *)malloc(x_size*y_size*sizeof(int));
  cgy=(int *)malloc(x_size*y_size*sizeof(int));

  for (i=5;i<y_size-5;i++)
    for (j=5;j<x_size-5;j++) {
        n=100;
        p=in + (i-3)*x_size + j - 1;
        cp=bp + in[i*x_size+j];

        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p);
        p+=x_size-3;

        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p);
        p+=x_size-5;

        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p);
        p+=x_size-6;

        n+=*(cp-*p++);
        n+=*(cp-*p++);
        n+=*(cp-*p);
```

```
ds.
     self      total
ls  ms/call   ms/call   name
 1   270.00    270.00   susan_corners
 3     0.00      0.00   getint
 1     0.00      0.00   _GLOBAL__I_usage_GCOV
 1     0.00      0.00   corner_draw
 1     0.00      0.00   get_image
 1     0.00      0.00   put_image
 1     0.00      0.00   setup_brightness_lut
```

hot spot

Which instructions implement this code efficiently?

Do not neglect compiler optimizations

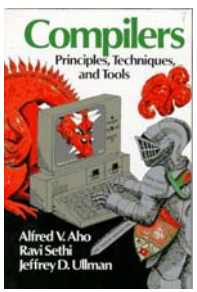


µ-profiling approach

```
372: 1447:  while(corner_list[n].info != 7)
  -: 1448:  {
371: 1449:    if (drawing_mode==0)
  -: 1450:    {
371: 1451:      p = in + (corner_list[n].y-1)*x_size + corner_list[n].x - 1;
371: 1452:      *p++=255; *p++=255; *p=255; p+=x_size-2;
371: 1453:      *p++=255; *p++=0;   *p=255; p+=x_size-2;
371: 1454:      *p++=255; *p++=255; *p=255;
371: 1455:      n++;
  -: 1456:    }
```

generate low-level C code

```
t4692 = (char *)corner_list_1011;
t4691 = n_1016 * 24;
t4690 = t4692 + t4691;
t4693 = (struct  lance_tag_13 *)t4690;
t4694 = (char *)t4693;
t4695 = t4694 + 4;
t4696 = (int *)t4695;
t4697 = *t4696 * x_size_1012;
t4698 = (char *)in_1010;
t4699 = t4697 * 1;
t4700 = t4698 + t4699;
t4701 = (unsigned char *)t4700;
t4704 = (char *)corner_list_1011;
t4703 = n_1016 * 24;
t4702 = t4704 + t4703;
t4705 = (struct  lance_tag_13 *)t4702;
t4706 = (char *)t4705;
t4707 = t4706 + 0;
t4708 = (int *)t4707;
t4709 = (char *)t4701;
t4710 = *t4708 * 1;
t4711 = t4709 + t4710;
t4712 = (unsigned char *)t4711;
p_1015 = t4712;
```



perform compiler optimizations

```
t6531 = (char *)in_1010;
t192 = (char *)corner_list_1011;
t6533 = 4 + t192;
t6535 = t6533 + t191;
t6536 = (int *)t6535;
t6537 = *t6536 * x_size_1012;
t6538 = t6531 + t6537;
t6541 = t192 + t191;
t6542 = (int *)t6541;
t6543 = t6538 + *t6542;
t6544 = (unsigned char *)t6543;
```

instrument code

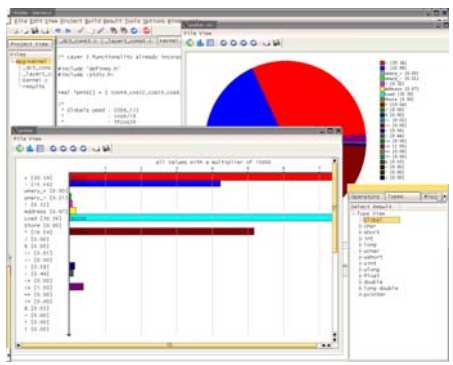
count ADD

count MUL

count LOAD

```
t6531 = (char *)in_1010;
t192 = (char *)corner_list_1011;
t6533 = 4 + t192;
t6535 = t6533 + t191;
t6536 = (int *)t6535;
t6537 = *t6536 * x_size_1012;
t6538 = t6531 + t6537;
t6541 = t192 + t191;
t6542 = (int *)t6541;
t6543 = t6538 + *t6542;
t6544 = (unsigned char *)t6543;
```

compile & execute

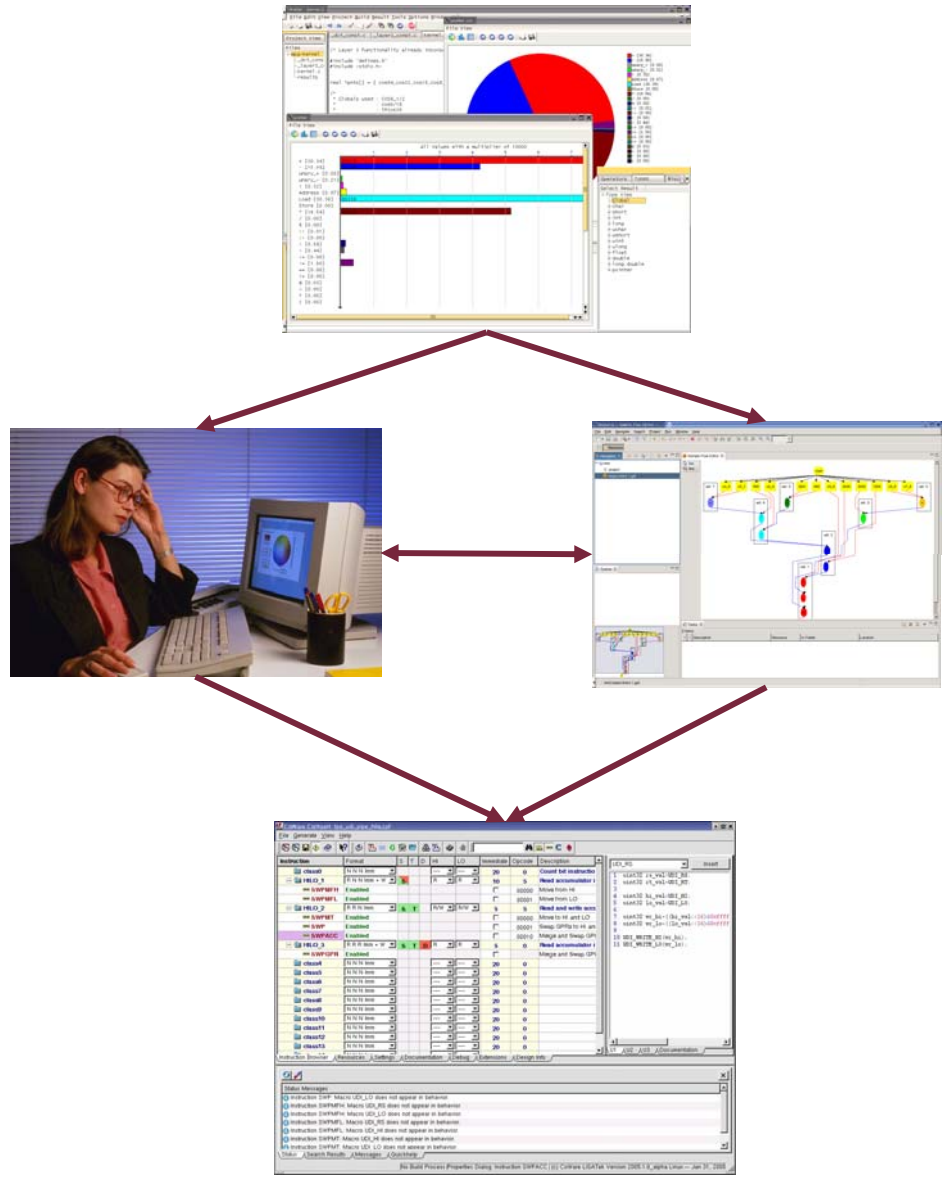


Profiler features summary

	C source level (e.g. gprof)	*assembly level (e.g. LISATek)*	*Micro-profiler*
primary application	Source code optimization	ISA and architecture optimization	ISA and architecture optimization
needs architectural details	No	Yes	No
speed	High	Low	Medium
Profiling granularity	coarse	fine	fine

Micro-profiler in the design flow

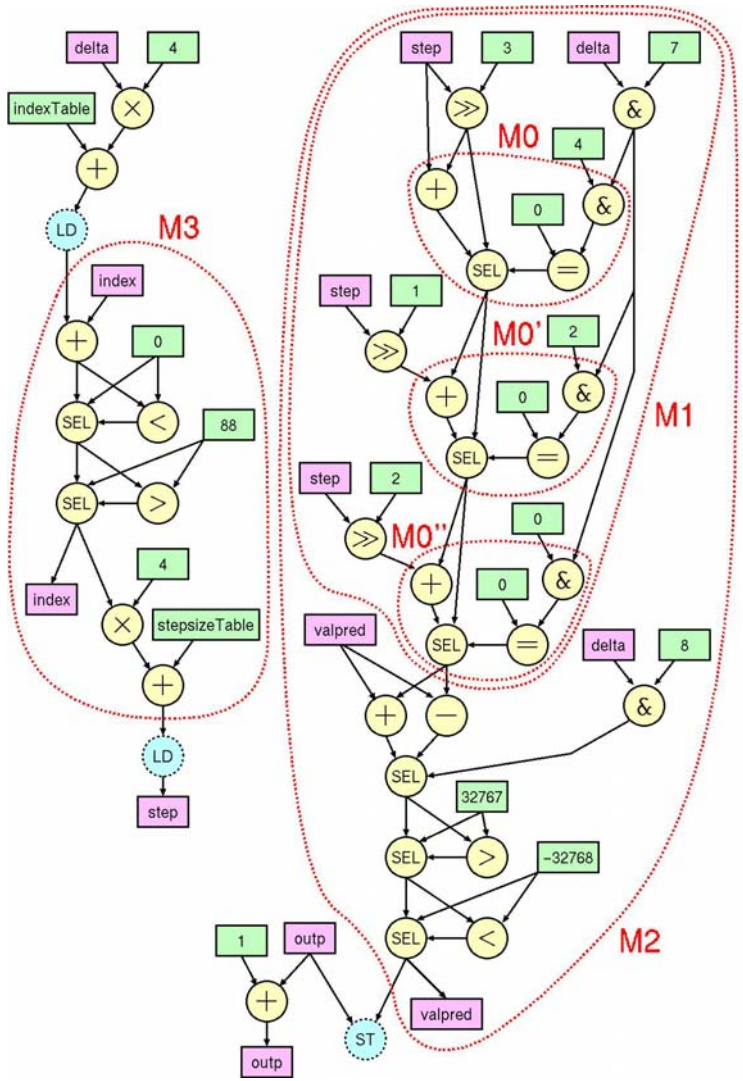
- Precise profiling of:
  - Operator execution count
  - Data type use
  - Variable word lengths
  - Memory access patterns
- Guide designer in basic architecture decisions:
  - Inclusion of dedicated function units (floating point, address gen.)
  - Leave out unused instructions
  - Optimal data path word lengths
  - Design of memory hierarchy (cache, scratchpad)
- Export profiling data for automatic ISA synthesis tool
  - Effects of compiler optimizations predicted early

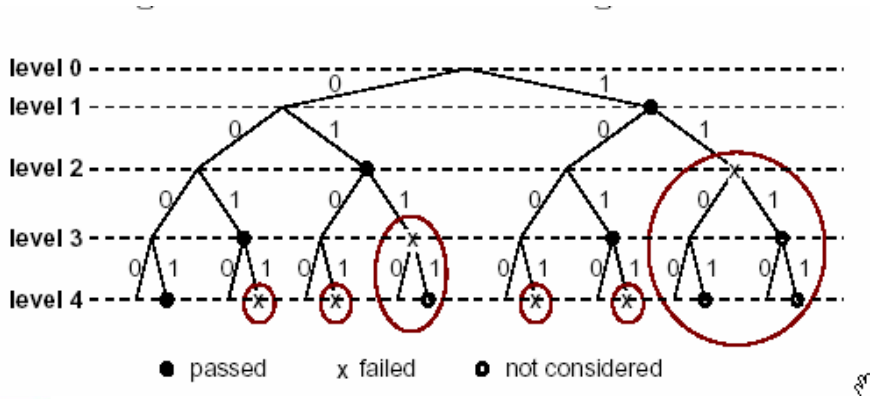


Custom instruction set synthesis: recent approaches

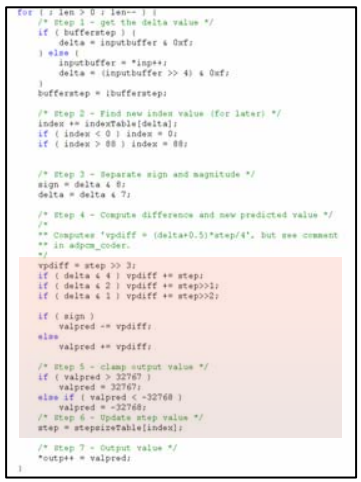
- J. Yang, C. Kyung et al.: MetaCore: *An Application Specific DSP Development System*, DAC 1998
- M. Gschwind: *Instruction Set Selection for ASIP Design*, CODES 1999
- K. Kücükcakar: *An ASIP Design Methodology for Embedded Systems*, CODES 1999
- H. Choi, C. Kyung et al.: *Synthesis of Application Specific Instructions for Embedded DSP Software*, IEEE Trans. CAD 1999
- F. Sun, S. Ravi et al.: *Synthesis of Custom Processors based on Extensible Platforms*, ICCAD 2002
- D. Goodwin, D. Petkov: Automatic Generation of Application Specific Processors, CASES 2003
- K. Atasu, L. Pozzi, P. Ienne: *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints*, DAC 2003
- N. Clark, H. Zhong, S. Mahlke: *Processor Acceleration through Automated Instruction Set Customization*, MICRO 2003
- ... and many others

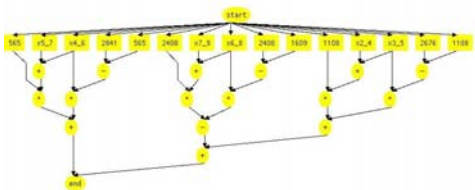
- Branch-and-Bound like algorithm for custom pattern identification under I/O constraints
- Capable of identifying optimal complex and disconnected patterns
- Requires fast and accurate cost estimation of speedup due to candidate patterns

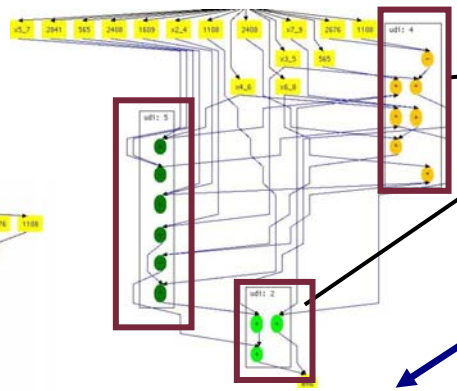




Custom ISA synthesis: ISS approach







optimal
custom instr

Performance estimation

Performance evaluation

Application code profiling (gprof, µprof)

Hot spot(s) graph model

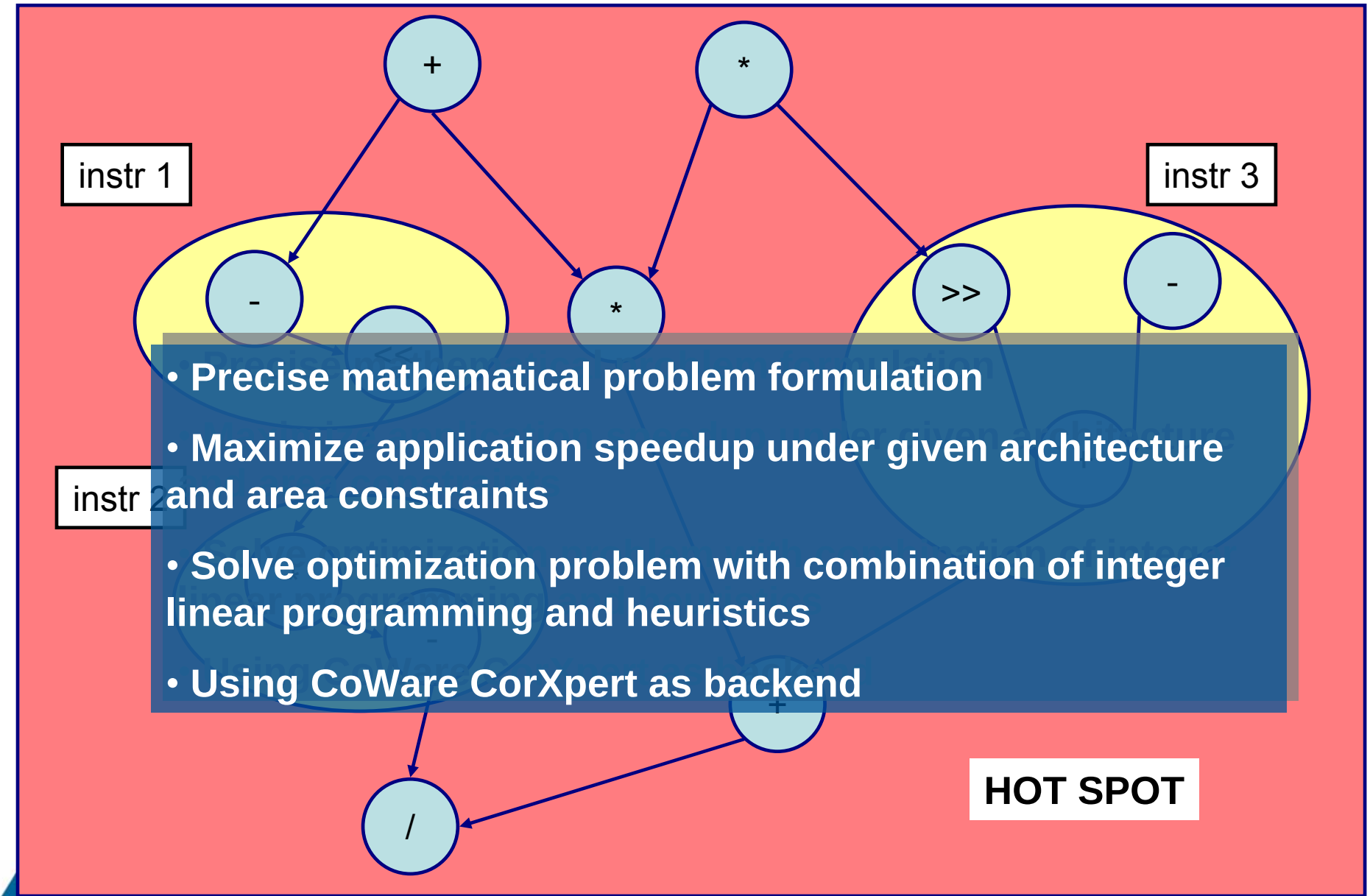
Semi-automatic ISA synthesis (GUI)

Processor generation (e.g. CorXpert)

User preferences

Architectural constraints

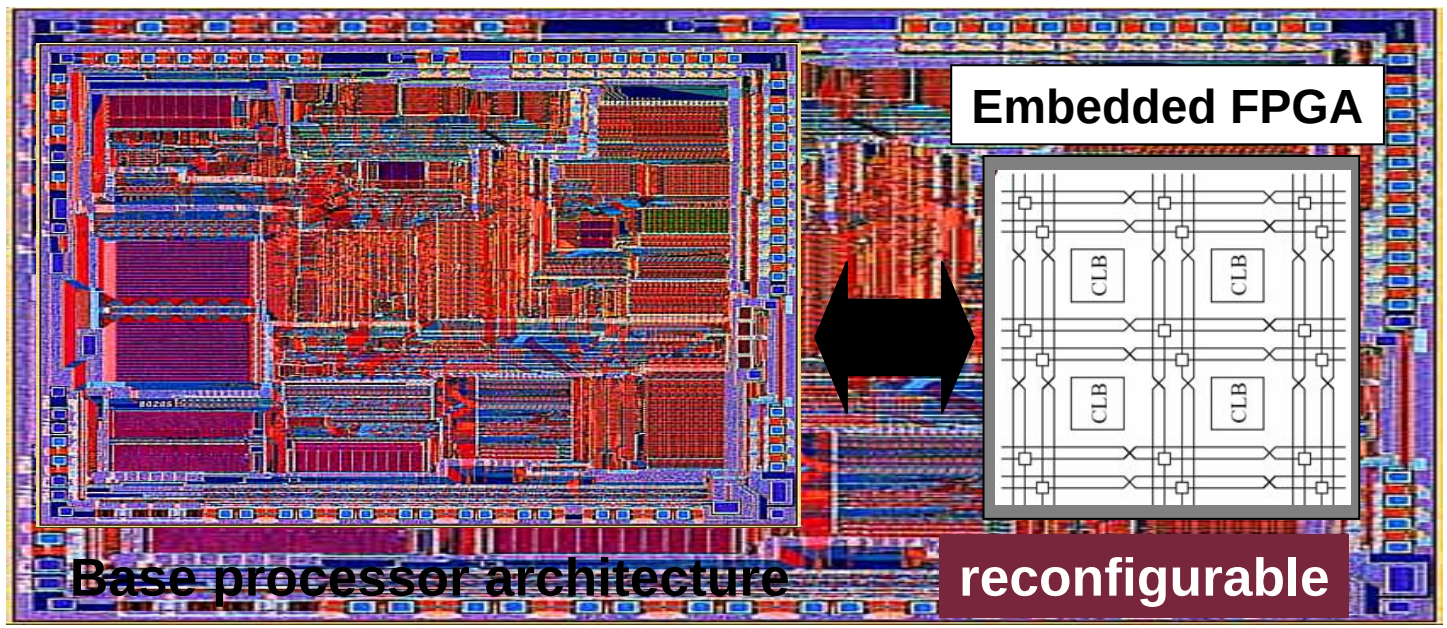
Abstract modeling: optimal data flow graph covering



- Precise mathematical problem formulation
- Maximize application speedup under given architecture and area constraints
- Solve optimization problem with combination of integer linear programming and heuristics
- Using CoWare CorXpert as backend

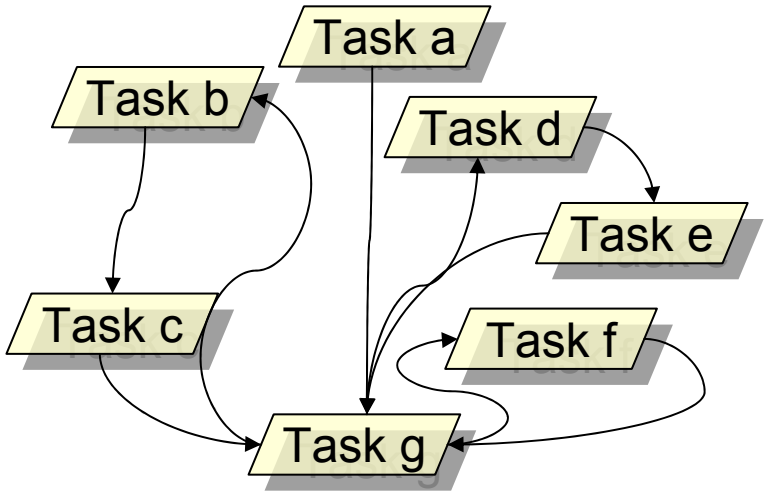
Outlook: reconfigurable ASIPs

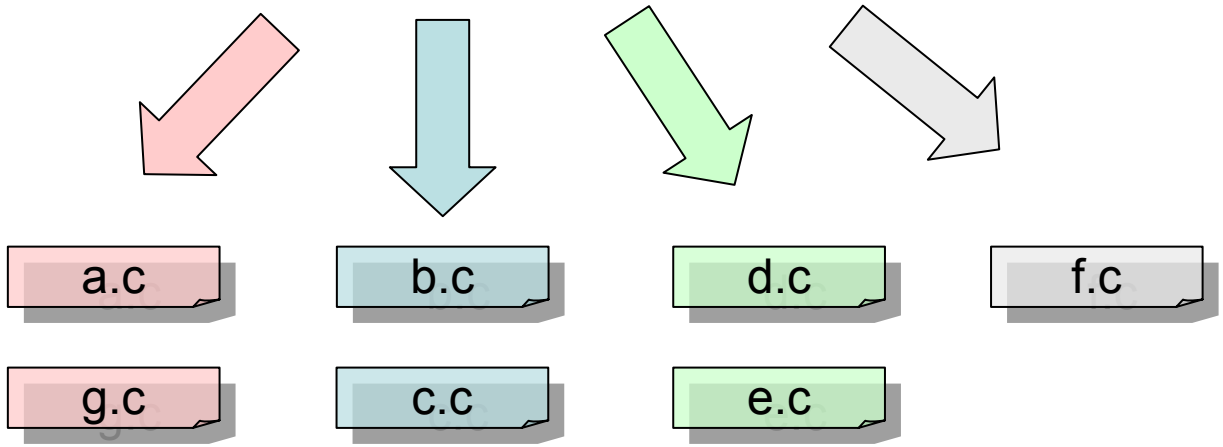
- Currently: (one-time) configurable ASIPs
- Combination of ASIP and FPGA:
  - “field-configurable ASIPs”
  - Fast adaptations w/o HW modifications

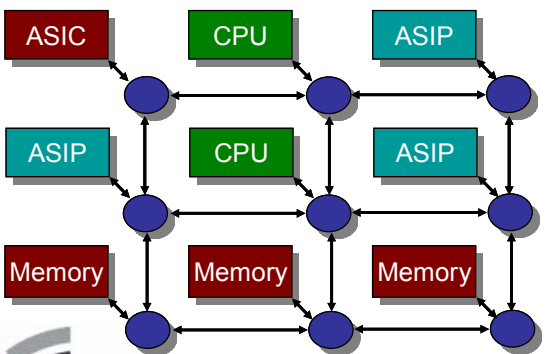


Outlook: compilation for heterogeneous MPSoC´s

- Urgently needed, but virtually not present today
- No tools even for simplest platforms (e.g. TI OMAP)
- Need for automated spatial and temporal task-to-processor mapping







References

- R. Leupers: *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*, Kluwer, 2000
- R. Leupers, P. Marwedel: *Retargetable Compiler Technology for Embedded Systems - Tools and Applications*, Kluwer, 2001
- A. Hoffmann, H. Meyr, R. Leupers: *Architecture Exploration for Embedded Processors with LISA*, Kluwer, 2002
- C. Rowen, S. Leibson: *Engineering the Complex SoC: Fast, Flexible Design with Configurable Processors*, Prentice Hall, 2004
- M. Gries, K. Keutzer, et al.: *Building ASIPs: The Mescal Methodology*, Springer, 2005
- P. Ienne, R. Leupers (eds.): *Customizable and Configurable Embedded Processor Cores*, Morgan Kaufmann, to appear 2006

Thank you !